OBJECT-ORIENTED ALLOCATION OF RESOURCES IN A

TACTICAL COMMUNICATIONS NETWORK

THESIS

Glenn R. Gier

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GE/ENG/88D-13

OBJECT-ORIENTED ALLOCATION OF RESOURCES IN A TACTICAL COMMUNICATIONS NETWORK

THESIS

Glenn R. Gier

AFIT/GE/ENG/88D-13

OBJECT-ORIENTED ALLOCATION OF RESOURCES IN A TACTICAL COMMUNICATIONS NETWORK

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Electrical Engineering

Glenn R. Gier, B.S.

Captain, USAF

December 1988

Preface

The Prototype Route Planner (PRP) described in this thesis is the indirect result of using object-oriented code to implement a routing algorithm for a tactical communications network. My original intent was to demonstrate that a routing algorithm could automate channel allocation, even though its host network is constrained by limited computing power at each node. Zenith 248's were to represent the nodes while their communications ports were to act as media links. Although PCs were readily available, none of them had the multiple ports necessary for a small network demonstration. Instead, I chose to simulate a network of multiple nodes and links on a single machine. To validate that the algorithm works even with a damaged network, I added a way to kill components. I believe that design concepts developed for PRP can be used to create tactical networks from scratch or to test how networks react when damaged.

Throughout this project, I received help and support from a variety of people whose contributions have become part of this work. I am deeply indebted to my advisor, Dr. F. M. Brown for his guidance and telepathic insight. Only he can find uncut gems amid the piles of rubble. Also my sponsor, Capt A. Morse, made possible initial travel funds and information that proved pivotal to the design of PRP. I am also grateful for the wonderful support provided by the men and women of MIT Lincoln Labs, especially Dr. H. M. Heggestad who made my extended visit possible and A. Steele who never tired answering questions about Flavors. Finally, I would like to thank my wife Joan for her encouragement and steadfast love.

Rick Gier

Table of Contents

List of Figures

AFIT/GE/ENG/88D-13

Abstract

The circuits in a military command and control network are expected to operate continuously in spite of changes and damage, and must be restored in minutes should they fail. This study examines circuit-building as a resource allocation problem, and describes an approach to the decentralized allocation of prioritized circuits in such a network. Based on a saturation search algorithm, a circuit-request message ripples from an originating node through the network to the circuit's destination node, providing a path exists. Along the way the message retains cost and path information that is used to attenuate expensive routes and provide path information during the back-sweep of the allocation phase.

An object-oriented program written in Scheme helped capture the details of nodes, links, channels, and circuits in the network, and showed how these components would interact when guided by the algorithm. The program has two purposes. First, it validates the concept that a common software-based assistant, distributed to each node, can aid operators in the rapid reconstruction of a badly damaged network. Second, it provides a planning aide for tactical network designers, who can use the program to model nodes, trunk-lines, channels and circuits.

A tactical network employing distributed allocation and priority-coding of its circuits, allows operators to forgo stored restoration plans as their principal means of maintaining service. This approach offers flexibility and responsiveness despite the likelihood of multiple outages and rapid changes, something that plans can not always deliver especially in time of war.

OBJECT ORIENTED ALLOCATION OF RESOURCES IN A TACTICAL COMMUNICATIONS NETWORK

I. Introduction

The Communications Nodal Control Element (CNCE) is a box-like structure the size of a step-van that contains switching equipment to support Air Force dedicated communications channels via microwave, satellite, and land-line media. Each deployable unit can be air-dropped or trucked to its field site, then activated to serve as a regional switching center, supporting data, voice lines, and other formats. These CNCE "nodes" permanently link field commanders to higher command authority, thereby ensuring combat theater communications essential for regional warfare (Warmuth 1988).

A typical deployment scenario weaves more than a half dozen CNCEs into a communications network. Positioned at strategic spots throughout a tactical theater, they provide a dedicated and redundant network of communications circuits. Associated with each scenario, an operations plan (annex K) details how communications personnel should interconnect adjacent stations to distant command authority, thereby forming the net. These links or trunks carry the dedicated communications circuits, while spare channels and currently-used, lower-priority channels in each trunk provide a vehicle for redundant backup. The unused trunk capacity also provides a means for communications personnel to accommodate additional circuit requests.

Problem Background

Depending on the environment, communications personnel face several kinds of resource allocation problems. The first deals with network planning. As each operations plan accommodates anticipated circumstances, diverse needs, and last minute changes, it is not

surprising that up to two months may pass before final acceptance of a plan by Wing or Numbered Air Force. Furthermore, during actual deployment the CNCE network must be fine-tuned to reflect new conditions. Technical Controllers presently revamp the network manually via patch cord connections, using protocols learned through war-time experience and peace-time exercises. The updates they make fall into allocative and restorative categories: (a) build dedicated circuits using channels to link CNCEs, or (b) re-route a segment of the circuit around a damaged link using spare or preempted channels (Warmuth, 1988).

Ironically, the patch cord feature that makes a CNCE network so very flexible imposes a labor-intensive process that detracts from responsiveness. The ultimate test occurs during war, when technical controllers assigned to each CNCE must reconstruct a severely damaged net without knowing the condition of the rest of the system. They will be constrained by a lack of global knowledge, as each CNCE controller knows only the status of channels connecting the specific CNCE to adjacent CNCEs. In order to effect repairs, each link's operability must be ascertained and communicated throughout the network. Although automatic switching equipment controls the dedicated lines once established, it is the communications personnel who must establish new requests and fix malfunctions using time-intensive manual techniques.

Overview

This study explores traditional algorithmic and constraint propagation techniques with the intent of implementing an object-oriented software solution for communications personnel to use to quickly establish dedicated circuits in a CNCE network. In Chapter II, the problem is explained in terms of an undirected graph, then the literature review of Chapter III covers previous research, placing special emphasis on approaches that work without global network knowledge. The design implementation of Chapter IV maps out an object-oriented approach that mirrors the environment, while Chapter V assesses the resulting Prototype Route Planner and projects the feasibility of

using a similarly designed Technical Controller's Assistant for use in an operational CNCE network.

Scope

Since building new communications circuits realistically includes alternative trade-offs, priorities, maximum hop (path) length, etc., this work focusses on ways to allocate communications channels in a pre-established network of trunk lines, given a request for new dedicated circuits. Since a damaged net is nothing more than a pre-established communication chain with broken links, these severed links form a special class of channel requests that when satisfied, also serve network reconstruction. For this reason, any automated approach that allocates channel resources for new circuits, also serves the dual purpose of patching damaged circuits.

Assumptions

This study assumes a small node population, as no more than 47 CNCE units were produced and only a portion of these will be connected for theater operations. CNCE units host modest computing power, (64K of RAM and limited permanent storage capacity); hence, this study presumes an external computing source, in each node, capable of implementing search procedures. Lastly, this study assumes that a CNCE node can talk to its neighbor using the same communications lines that carry data. Inability to contact a neighboring node, even if caused by a malfunction, is interpreted as destruction of that node.

II. Problem Analysis

Some problems can be resolved by decomposing the whole into individually solvable pieces. Object oriented programming adheres to this philosophy in that it breaks a system into objects that function like their real-world counterparts, and in doing so permits the programmer to concentrate on one well understood object at a time (Booch, 1987:16). This chapter paves the way for an object oriented design by presenting a CNCE network as a graph made of node, link, and channel objects. It then reviews how to allocate channels to a circuit, and concludes with manual circuit restoration techniques.

A CNCE tactical communications network exists to carry messages from one destination to another; this is accomplished through a web of interconnected CNCEs and trunk lines. The circuits in a CNCE network are bi-directional; hence an undirected graph conveniently describes the major physical features of such a network. The circular nodes of an undirected graph represent the Communications Nodal Control Elements, while the links (connecting lines) of an undirected graph depict the trunks that carry communications circuits. Throughout this paper the term "circuit" refers not to the graph-theory definition – a series of links that begin and end at the same node – but to a telecommunications connotation, where two parties talk through a path that does not double back on itself. In graph-theory nomenclature, this form of "circuit" is actually a simple (each link used only once), elementary (each node used only once), chain of links connecting one node to another (Christofides, 1975:4).

Network Representation

Figure 1 shows how the CNCEs and trunks of a simple network appear in graph form. The nodes (CNCEs) serve as junctions, terminating the connecting links (trunks) of the network. The links span only a single pair of nodes, and unlike a node, which can exist without links, a link must adjoin nodes (typically two distinct nodes). Although links carry circuits, there exits a unit of

conveyance that is more basic than the trunk. This property belongs to channels which propagate the message contents of a single circuit and which are characterized by some transmission rate, 64-kbps for example. A trunk typically contains 172 channels that are time-sequenced (Sues, 1988). In Figure 1, a blow-up of one of the representative links shows that it contains six channels. One shortcoming of the undirected graph representation so far is that it fails to show the used and unused channels of a trunk. Normally, trunks contain numerous channels, any one of which can

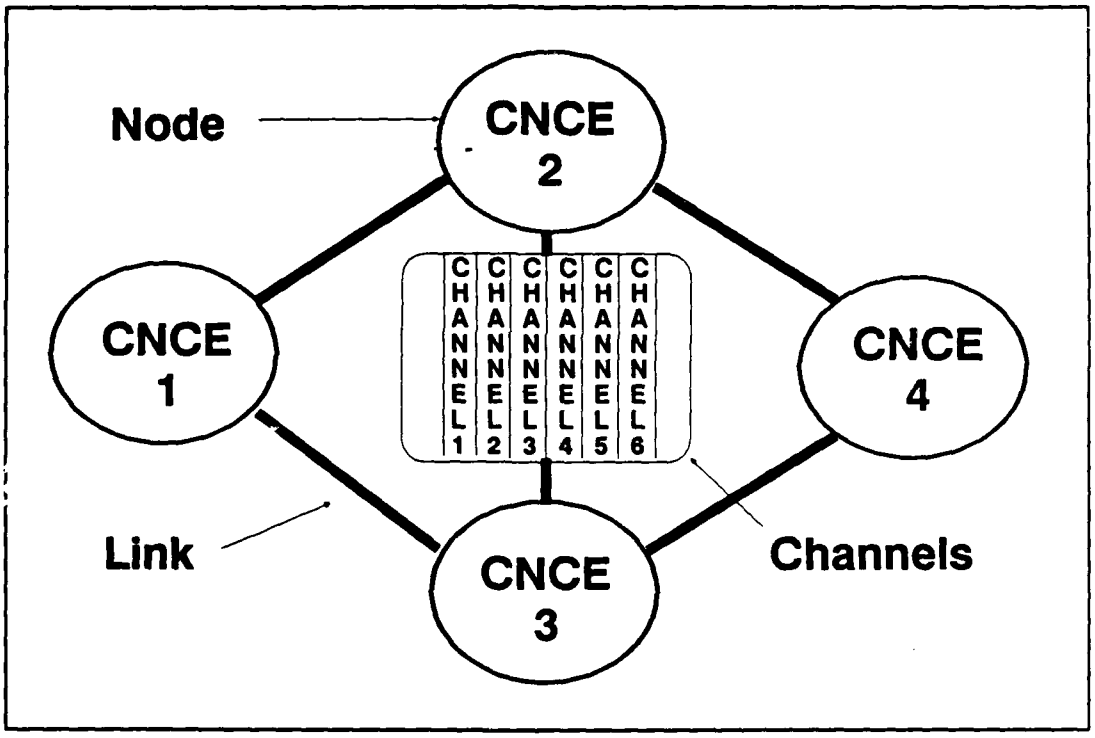

Figure 1. CNCE Network

be allocated to a circuit to pass information between CNCEs (in reality some channels have higher transmission rates than others, but for the purpose of this study, it is assumed that all channels have identical throughput and are interchangeable). The number of channels in a trunk determines

its capacity. One could argue that channels themselves should be represented by links, but this convention ignores the trunk as an entity and clutters the graph with numerous parallel lines. For this reason, the following graphs use only nodes for CNCEs and lines for trunks.

Since channels have no pictorial representation, the following labeling scheme explicitly shows the capacity of each link and affords the observer a better idea of the capacity of the overall network. First, let us describe the network as $\{ CNCE_1, \ldots, CNCE_n \}$, the set of CNCE nodes in the tactical network. Next, let $Link_{ijk}$ denote a trunk (if one exists) connecting nodes $CNCE_i$ and $CNCE_j$, where k distinguishes parallel links when they exist. Finally, let q_{ijk} be the capacity of $Link_{ijk}$, expressed so as to show the number of circuits, by priority, that the link can support. This last requirement sounds complicated, but if one uses vector notation to display a sort of prioritized-capacity, an entire network can be described (figure 2).

Before proceeding further, one should note that if this were not a military network, there would be no need to represent capacity as a vector. An equivalent no-priority network needs only a single number to relate its capacity since there are only allocated and unallocated channels. Indeed most graphs, including those in the next chapter, use this convention.

The CNCE tactical network recognizes a circuit hierarchy based on priority. This study arbitrarily employs three priorities: A, the highest; B, an intermediate; and C, the lowest priority. These correspond to priorities in a tactical network. A circuit's importance determines its priority, and when carried via the network, all participating channels assume the same priority. Prior to allocation, a channel has no status and is considered spare. Looking at all the channels in a trunk, one can summarize their use and capacity with the vector [A B C spare]. This four-place vector shows the number of channels allocated to A, B, or C priority circuits as well as the number of channels in spare status. In addition to displaying the present state of a trunk's channels, this vector provides additional information about its ability to support more circuits. For example in figure 2, consider $q_{231} = [0\ 2\ 1\ 3]$ which says that $Link_{231}$ carries no A-priority circuits, two

B-priority circuits, one C-priority circuit, and three spare channels for a total of six channels, three of which are allocated.

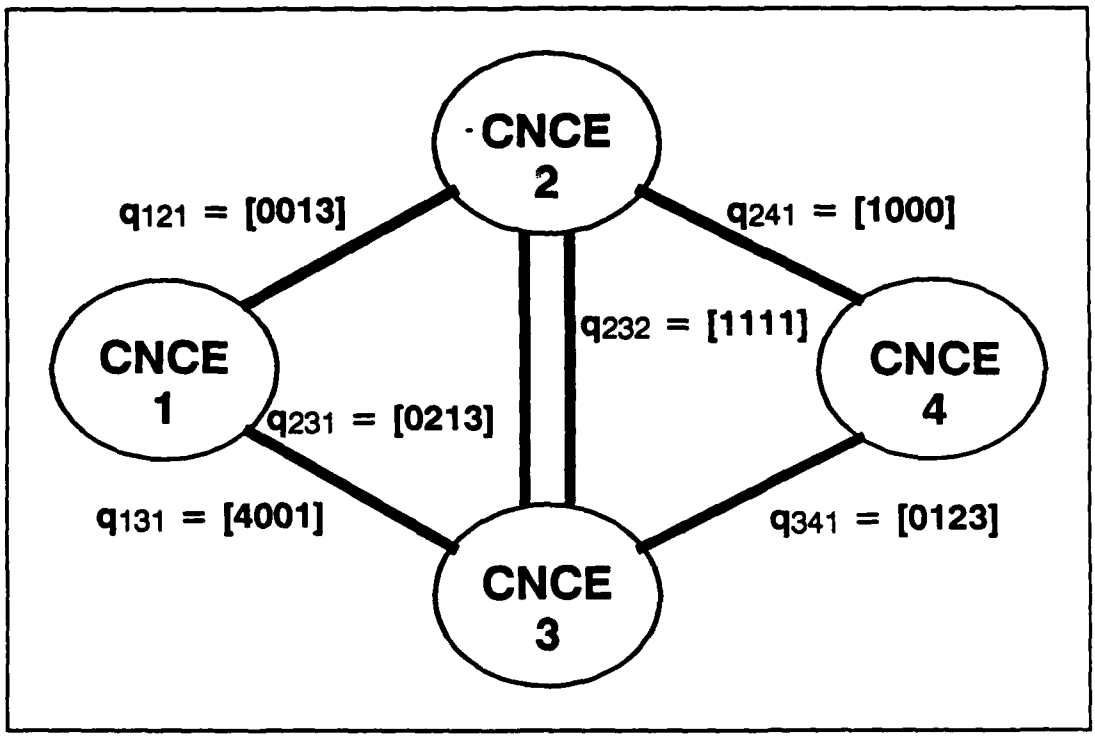

Figure 2. Example Network Graphical Representation

Allocation Considerations

A communications person called a technical controller could use this information to allocate a channel to a new circuit. Given a B-priority circuit request, the technical controller could pick any of the three spares or the C-priority channel to dedicate to the new circuit. The technical controller can not preempt B-priority channels since they have the same priority as the request. Likewise, if there were any A-priority channels, the technical controller could not preempt a higher priority channel for a lower priority request.

The same vector notation can be used to count the cost of establishing a new circuit. Intuitively, one can see that picking a spare channel in the above example will not disrupt any active circuits. But if the C-priority channel is preempted to fill the new request, the network suffers the penalty of a broken circuit. One way to measure the cost of honoring a circuit request is to think of it in terms of a cost vector. Let c_{ijk} = [A B C spare] , represent the vector cost of using a channel in $link_{ijk}$. The value of c_{ijk} depends on the precedence of the circuit using the channel-resource. If unallocated, the channel costs one spare, or [0 0 0 1] , to use. If previously allocated, the priority of the owning circuit dictates the cost of preempting the channel (for example, a top priority channel costs c_{ijk} = [1 0 0 0] to use). With this framework, one can construct a circuit or path which connects an origin-node to a destination-node via a route of alternating links and support nodes, whose total cost is the sum of all the c_{ijk}s produced along the way. Thus a circuit's cost might look like [0 0 1 3] , which says that one C-channel and three spare channels are needed to build the proposed circuit.

The technical controller's task is to route a circuit from an originating CNCE, through the network, terminating at the destination, without exceeding trunk capacity, at minimum cost, and without knowing in advance current channel allocations. Prepared backup plans simplify this task for the technical controller. If a circuit goes bad, the controller switches to the backup circuit, and if time permits, isolates the bad segment for later repair. If the fault is isolated to a bad channel, swapping it with a spare channel rejuvenates the now inactive circuit for subsequent backup use. This method of circuit maintenance works well for repairing infrequent malfunctions, but a different mode of operation prevails during massive outages. In the event of significant damage, technical controllers restore service based on priority. First they fix high priority circuits using available backups, but if there are none, they reinstate service at the expense of lower priority circuits. Depending on the circumstances, the repair process can quickly exceed the limits of a backup plan. For this reason the next chapter considers algorithmic approaches to alleviate potential

problems during massive outages, while the chapter after it shows how to apply object oriented programing to model the nodes, links, and channels just discussed.

III. Literature Review

The intent of this literature review is to find a suitable algorithm that can be used to find circuit routes within a damaged CNCE network. As much work has already been done on routing and search algorithms, it is only logical to assume that some previously applied solution can be adapted to work in the tactical communications environment. Routing and search algorithms are found imbedded in many areas, not just in communications, therefore this review analyzes: a) conventional algorithms, b) future communications protocols, and c) constraint-based approaches. The key concepts used in each methodology can be applied to the design of a distributive allocation algorithm for a CNCE network.

Conventional Algorithms

Operations research books and journals abound with discussions of conventional algorithms used to optimize resources. They afford a good starting point for CNCE network channel allocation, as many such algorithms were originally designed for computer optimization of resources in a network. Three algorithms in particular – the maximum flow algorithm, the minimum cost flow algorithm, and the shortest path algorithm – share linear programing as a common ancestor, and offer valuable insight into the CNCE network problem. With each of these algorithms, be aware that they assume a network has a single source and a single sink. When comparing CNCE features to these algorithms, think of a circuit's originating node as its source, the destination node as its sink, and the circuit as needing one channel resource every time it advances from one node to the next.

Linear Programming. At first glance, constructing a network made up of CNCEs appears to be a task best done using operations research techniques. If the goal is to maximize communications throughput, a linear programming approach will maximize the flow out of a source node (or the flow into a sink node), given the capacity of each arc connecting the network.

Although a detailed explanation exceeds the scope of this section, a quick look at the variables in the linear programming model shows a parallel to other algorithmic approaches. The object of linear programming is to choose values of independent variables so that they maximize or minimize some overall objective function. Constraints limit each variable's range of values. A linear programming model proceeds step-wise, improving the objective until reaching an optimum or infeasible solution. But one necessary condition for linear programming is total knowledge of the network status before optimization. Even when given this information, the full power of linear programming is not necessary for efficient resource allocation.

Maximum Flow Algorithm. Markland demonstrates an intuitive and efficient method of network maximization that is simpler than a linear program. Whereas before q_{ijk} indicated the capacity of a single link, Markland uses the symbol c^*_{ij} to represent the capacity for flow in a complete path, from nodes i to j. The asterisk denotes optimality since the path as a whole can carry only as much as its most restrictive segment will permit.

1. Find a path from source to sink with positive flow capacity (*each path segment must* support the conveyance of at least one unit of flow from source to sink. Sink to source flow is negative). Obviously, if none exists, the net flows already assigned constitute a maximal flow pattern.
2. Search this path for the branch with the smallest flow capacity. Denote this capacity as c^*_{ij}, and increase the flow in this path by c^*_{ij}.
3. Decrease by c^*_{ij} the flow capacity of each branch in the selected path.
4. Increase by c^*_{ij}, in the opposite direction, the flow capacity of each branch in the selected path.
5. Return to step 1 and repeat the procedure outlined in steps 2, 3, and 4 until no paths with positive flow capacity remain.
6. Compute the net flow in all branches for which flow(s) have been assigned in both directions (Markland 1983:388).

The above method optimizes a network, but only after an exhaustive search of all paths, from source to sink, fails to reveal unused capacity. Clearly, if the maximum capacity of a network were known ahead of time, one could stop searching when the algorithm's new-found capacity equals the theoretical maximum. The Max Flow – Min Cut Theorem conveniently achieves just that:

> If, for any network, we find the cut value for each of the finite number of cuts that can be made in the network, then the smallest total capacity (cut value) is equal to the maximum flow in the network (Markland, 1983:392).

Thus, one can maximize flow in a CNCE communications network, with a single source and sink, using the above algorithm, given global knowledge of node capacity and arc assignments—making it extremely useful for operations planning—with the understanding that it will not necessarily find the shortest path but will find the maximum flow.

Minimum Cost Flow Algorithm. The minimum cost flow algorithm appears similar to the maximum flow algorithm, yet it adds the constraint of finding the shortest available path within the capacity confines of the network. This algorithm accurately describes the CNCE network channel allocation process with the caveat that it does not implement preemption of already assigned channels. In the following equations s represents a single source node, t a single sink node, i and j intermediate nodes, f_{ij} is the arc flow from node i to node j, c_{ij} is the cost of

$$\sum_j (f_{sj} - f_{js}) = v$$

traveling the arc between nodes i and j, and q_{ij} is the capacity of the arc connecting nodes i and j. The relation

$$\sum_j (f_{tj} - f_{jt}) = -v$$

signifies that the arcs connected directly to the source must absorb the source's output, v.

$$\sum_j (f_{ij} - f_{ji}) = 0 \qquad (\text{for all } i \neq s, i \neq t)$$

Likewise

represents the flow discharged by each arc into the sink. The relation

$$0 \leq f_{ij} \leq q_{ij}$$

is the law of conservation that says that the net flow out of (in to) any node that is not a source or

sink is zero, and

$$\min \sum_{(i,j)} c_{ij} \; f_{ij}$$

relates that for all intermediate nodes, the flow through an arc must be positive and must not exceed the arc's capacity. The goal is to minimize the cost as shown by subject to the capacity constraints of the arcs and the volume of flow from the source. This can be accomplished by summing the cost-quantity products for every arc in the network. Keep in mind that the c_{ij}s are integers and in a CNCE network, would all cost the same.

> The minimum cost flow algorithm first sends as many units from s (source) to t (sink) that incur a total cost of 0 each for the entire journey from s to t. Next, the minimum cost flow algorithm sends as many units as possible from s to t that incur a total incremental cost of 1 each for the entire journey from s to t, etc. The algorithm stops when a total of v units have been sent from s to t, or when no more units can be sent from s to t, whichever happens first (Minieka, 1978:107).

To keep track of the incremental or augmenting costs during the algorithm, each node owns a cost counter that tracks the cost of flow in the forward direction and flow in the reverse direction (reverse flow facilitates the augmenting process without exceeding branch capacity constraints). If two adjacent nodes have different cost counter values, their difference represents the cost of flow between the nodes, or cij.

A summary of the minimum cost flow algorithm explained by Mineka proceeds as follows:

1. Initialize the flow of each arc to zero and set the cost counter of every node to 0.
2. Decide which arcs can have flow changes by examining the cost potential across the arc, its capacity, and those arcs already carrying flow.
3. Use the max flow algorithm to increment each cost counter and to find each path request. If there is a path for every unit of source flow, stop as you have the lowest cost using those paths. If not all requests can be honored proceed to step 4.
4. Increase the price of payment to complete the path, do not consider those arcs with full capacity, and return to step 2.

The algorithm finds a path for every source unit or until the network is saturated. This approach would work well for a network overrun by new circuit requests as it gets the most circuits through using the shortest distances. The algorithm is not adapted for priorities however.

Dijkstra Shortest Path Algorithm. Dijkstra's Shortest path algorithm uses temporary and permanent labels to determine the shortest path in a network. The algorithm is as follows:

1. A permanent label of zero is assigned to the source node. All other nodes are assigned temporary labels that are set equal to the direct distances from the source node to the nodes being examined. If any node cannot be reached directly from the source node it is assigned a temporary label of +infinity.
2. All of the nodes having temporary labels are examined and the node having the minimum of the temporary labels is selected and declared permanent. If there are ties between temporary labels, break the tie arbitrarily.
3. Suppose that node T has been assigned a permanent label most recently. The remaining nodes with temporary labels are examined, by comparing, one at a time, the temporary label of each node to the sum of the permanent label of node T and the direct distance from node T to the node being examined. The minimum of these two distances is assigned as the new temporary label for that node. Note that if the old temporary label for the node being examined is still minimal, then it will remain unchanged during this step.
4. Now select the minimum of the temporary labels and declare it permanent. If there are ties, select one, but only one, and declare it permanent. If the node just declared permanent is the sink node, the algorithm terminates and the shortest route has been determined. Otherwise return to Step 3 (Markland 1983:393).

After the algorithm terminates, the shortest path is identified by retracing the path backward from the sink node to the source node, selecting the nodes that were permanently labeled at each step. Although the shortest route algorithm will find the optimum route in a communication network when each link is assigned a cost or path equal to one, it does require global knowledge at steps three and four, to decide which of the permanent and temporary nodes to pick for subsequent inspection.

DPAS Network Control System Algorithms

The newest Air Force switching network, the digital patch and access system (DPAS), shares many similarities with a CNCE tactical communications network. As in the CNCE model, the DPAS nodes serve as junctions for channels. In addition, the DPAS network control system (DNCS) adds a control hierarchy to maintain the net. Just as technical controllers in a CNCE network isolate bad circuits and activate backups on a tactical level, the DNCS will accomplish the same function automatically on a strategic level through the use of automated test equipment, digital switching networks and computers. The first stage of DPAS implementation requires manual system control via "electronic patch cords", while later plans call for an automatic mode, essentially freeing the technical controller for other work (Computer Sciences Corporation 1988:3-1). As the DNCS provides system "smarts", it must accomplish the following:

- Respond to the arrival of control messages
- Sense failures
- Find routes
- Connect circuits via new routes, preempting other circuits as necessary
- Release all segments of failed or preempted circuits to spare status, thus providing network resources to support attempts to restore failed or preempted circuits
- Cancel connection attempts if they cannot be completed so as to prevent partial circuits from remaining up
- Confirm all connections and releases and document them in link fill tables that form a real-time distributed connectivity data base (Computer Sciences Corporation, 1988:7-9)

The Technical Controller in a CNCE network currently accomplishes all the above duties. The intent of this study is to look at ways to help the controller with route planning. In this respect, a CNCE route planning algorithm can benefit from DNCS algorithm research. Three algorithms, discussed in the following paragraphs, have been considered for DNCS route planning. Algorithms 1 and 3 show considerable promise; Algorithm 2 has been eliminated due to the high message traffic it generated. Algorithm 1 is similar to Ludwig and Roy's call-routing algorithm (Ludwig and Roy, 1977:1353), while Algorithm 3 is an adaptation of the shortest path algorithm.

Algorithm 1. The destination node, in the DNCS version of Algorithm 1, initiates a SEARCH message whenever it needs to find a route to the origin node. In doing so it sends the SEARCH message to all adjacent nodes that attach via spare or lower priority channels. In other words, if two nodes connect only through fully committed, top priority channels, and the SEARCH message seeks out a low priority route, then the SEARCH message will never transit the top priority link as it can not bump a channel or find a spare. As the SEARCH messages filter through the network, they collect costs based on hop–length and priority. Eventually (if a path exists), a message reaches the source node, where it starts a timer and waits for other SEARCH messages to arrive. As the messages arrive, the source node orders the routes by increasing cost, displaying the results for the controller to act on. After the time expires, the source node uses the lowest cost or most appropriate route information to retrace its steps, telling the supporting nodes to commit resources for the chosen path. Thus, each node stores only information about itself, its neighbors, and the status of the links leading to its neighbor nodes. In addition, each node maintains a search-in-progress table that tracks a fixed number of lowest-cost requests (Computer Sciences Corporation 1988:7-11).

Algorithm 3. Algorithm 3 uses a form of the shortest path algorithm discussed earlier. The modified version works around the global knowledge requirement by storing link allocation information in a table for every link in the network (Computer Sciences Corporation, 1988:7-14). CHANGE messages, dated with time-stamps, update each node's table whenever a channel status changes. Due to propagation delays, one part of the network may not know the status of another part of the network. By examining each CHANGE message time stamp and rejecting old information for new, the node learns about the global status of link assignment.

Between Algorithm 1 and Algorithm 3, the first appears the better suited for CNCE implementation. The algorithm's simple design and limited need for internal storage of intermediate results make it particularly attractive in the memory–constrained environment of a CNCE network. Algorithm 1 passes fewer messages and thereby places less of a demand on the

network. On the other hand, Algorithm 3 will always optimize a route, but the end does not justify the means—from a practical viewpoint, a working circuit is all that is needed. One common theme is that both algorithms use the same structure to describe the channel allocation process discussed next.

DNCS Link Costing Scheme. Computer Sciences Corporation outlines a way to determine link cost in a preemptive environment, with obvious implications for a CNCE network. In a military network employing circuit restoration priorities, the "best route" corresponds to the lowest cost route, where the route cost parameter measures both the efficient use of network resources and the number and importance of subscribers who may be denied service if it

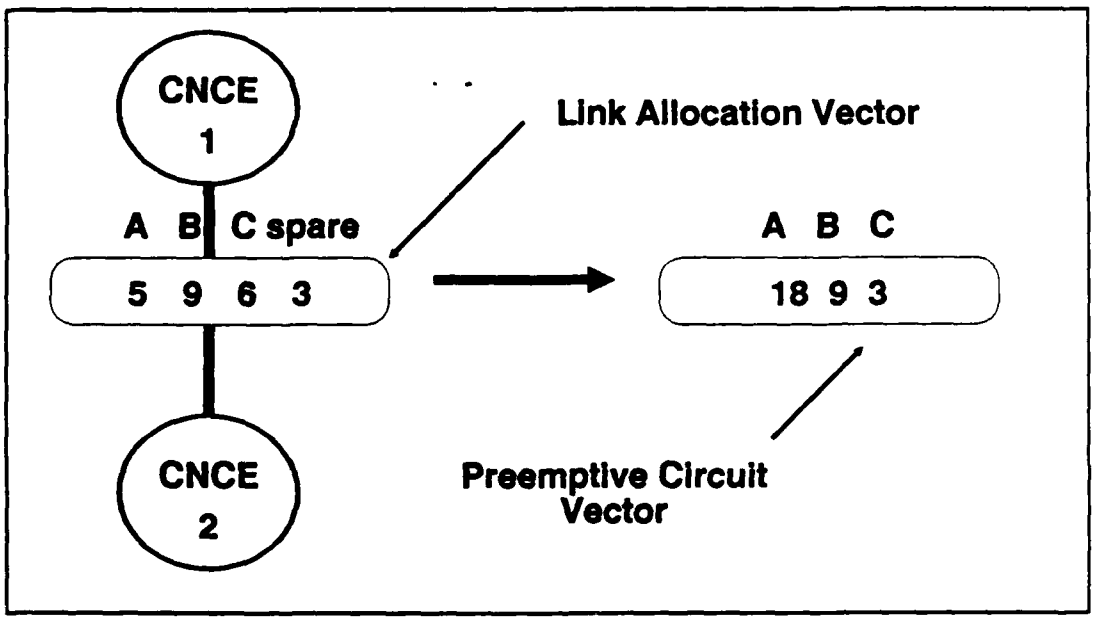

Figure 3. Link Costing Scheme

becomes necessary to preempt a channel to restore a higher priority circuit (Computer Sciences Corporation, 1988:7-15). Thus, the costing scheme is designed to serve the maximum number of

high priority users. Potentially valuable as a CNCE protocol, the DNCS costing scheme formalizes the priority, capacity, and availability of links that connect two adjacent nodes, or digroups (digital groups). Consider a north and south node connected by a link made of many channels (Figure 3). Each channel owns a priority attribute (arbitrarily A, B, or C for this study) if it participates in a dedicated circuit; otherwise an unassigned channel retains its spare status. A link allocation vector describes the overall channel composition (the same as q_{ij} in the previous chapter). Ordered by decreasing priority, the allocation vector contains the number of channels assigned to each priority, as well as those in spare status. For example, the number 5 signifies five channels hold the highest priority while the last number, 3, represents three spares. By modifying this format slightly, one can easily build a preemptive circuit vector that shows, by priority, the number of channels capable of fulfilling a request. First, discard the number of top-priority channels since they can not be preempted even by another A-priority request. Then replace each allocation vector component with the cumulative sums of all lower-priority channels.

The preemptive vector quickly identifies the quantity of channels available to any priority circuit. The vector, however, does not specify which resource will be used first (spare versus lower priority). Thus the algorithm designer faces a preemption choice, pitting cost against the value of an established connection.

Constraint-Based Approaches

This study's final source of algorithmic information comes from specific applications within the artificial intelligence community. The applications themselves include pattern recognition and distributive problem-solving, but more importantly, the tools used to solve these problems can also be applied to allocating channels in a tactical communications network. Rather than apply constraints to centrally collected data, as in linear programming, constraint propagation instead filters data-nodes through a sieve made up of constraints that capture some aspect of the problem domain. The constraints eliminate a portion of the solution space that

contains infeasible solutions, thereby reducing the time needed to search the remaining feasible space for a proper solution. Constraint problem-solving uses a two-step process. First, analyze the problem domain to determine what the constraints are. Second, solve the problem by applying a constraint satisfaction algorithm that effectively uses the constraints from the first step to control the search (Rich, 1983:351).

Before going into more detail about identifying constraints and how to control search, a look into the early uses of constraints will serve as a springboard for later discussion. Although not the first to use constraint propagation, Stallman and Sussman used this term as well as dependency directed backtracking to describe conditions in a computer-aided circuit analysis expert system called EL, which was written in a problem–solving rule–based language called ARS. In the EL context, constraint propagation refers to symbolic, linear relations attributed to certain aspects of electrical components (Stallman and Sussman, 1977:138). It takes the form of demons that add constrained assertion values to the data base whenever their antecedent conditions are met. These changes provoke other demons to fire until no new assertions are added, or a contradiction appears. The term dependency directed backtracking describes how the system deals with contradictions.

> The antecedents of the contradictory facts are scanned to find which nonlinear device state guesses (more generally, the backtrackable choicepoints) are relevant; ARS never tries that combination of guesses again. A short list of relevant choicepoints eliminates from consideration a large number of combinations of answers to all the other (irrelevant) choices (Stallman and Sussman, 1977:138).

The importance here is that small changes in the value of a parameter do not necessitate update of the entire data base, and when conflict does arise, major portions of the search space do not have to be re-examined.

Waltz Algorithm. Constraints have been applied to other areas in other ways. Waltz used constraint satisfaction to deal with the problem of perception. His algorithm labels corners in a blocks world scene and deduces from these constraints the nature of objects (Rich, 1983:351). Rich summarizes the corner finding process of the algorithm:

1. Find the lines at the scene/boundary border and label them. These lines can be found by finding an outline such that no vertices are outside it.
2. Number the vertices of the figure to be analyzed. These numbers will correspond to the order in which the vertices will be visited during the labeling process. To decide on a numbering, do the following:
    1. Start at a vertex at the boundary of the figure. Since boundary lines are known, the vertices involving them are more highly constrained than are interior ones.
    2. Move from the vertex along the boundary to an adjacent unnumbered vertex and continue until all boundary vertices have been numbered.
    3. Number interior vertices by moving from a numbered vertex to some adjacent unnumbered one. By always labeling a vertex next to one that has already been labeled, maximum use can be made of the constraints.
3. Visit each vertex V in order and attempt to label it by doing the following:
    1. Using the set of [eighteen] possible vertex labelings ... attach to V a list of its possible labelings.
    2. See whether some of these labelings can be eliminated on the basis of local constraints. To do this, examine each vertex A that is adjacent to V and that has already been visited. Check to see that for each proposed labeling for V, there is a way to label the line between V and A in such a way that at least one of the labelings listed for A is still possible. Eliminate from V's list any labeling for which this is not the case.
    3. Use the set of labelings just attached to V to constrain the labelings at vertices adjacent to V. For each vertex A that was visited in the last step, do the following:
        1. Eliminate all labelings of A that are not consistent with at least one labeling of V.
        2. If any labelings were eliminated, continue constraint propagation by examining the vertices adjacent to A and checking for consistency with the restricted set of labelings now attached to A.
        3. Continue to propagate until there are no adjacent labeled vertices or until there is no change made to the existing set of labelings (Rich, 1983:357).

Constraint Satisfaction. Constraint satisfaction differs from the previous constraint propagation example in that satisfaction uses a list of labels to annotate a set of constraints, whereas propagation conveys an explicit linear relationship. The advantage to Waltz's algorithm is that it guarantees a correct figure labeling if one exists. Furthermore, its utility increases as the ratio of constraints to nodes increases (Rich, 1983:358). Davis notes that the Waltz algorithm is always complete for a single constraint, but that it is not always complete for a set of constraints (Davis, 1987:290).

Even though differences exist between label and linear versions of constraint propagation, Davis concludes that they share valuable common properties:

- Simple, easy to update control structures
- Yield partial answers under time limitations
- Easily implemented in parallel
- Well suited to incremental systems
- Work well for localized information (Davis, 1977:283)

The upshot of these similarities is that the CNCE network problem falls into a class of constraint labeling problems that can be solved efficiently. Davis likens a constraint label system of inequalities to the solution of a shortest path problem on weighted graphs, and finds that:

> The Waltz algorithm is complete for the whole inference process (assimilation together with query answering). Moreover, if we perform refinement in the proper order, then, for consistent sets of constraints, the system reaches quiescence in time $O(n^3)$ (Davis, 1987:304).

By assimilation, Davis means the process of finding the path from the originator to the destination. Query answering is the return of the optimal answer from the destination to the originator. Not surprisingly, this particular instance of constraint propagation captures elements of the Minimum Cost Flow algorithm presented earlier. The Waltz algorithm merely details how to go about building the path and how to pick the best path during refinement.

Multistage Negotiation in Distributed Planning. Conry, Meyer, and Lesser describe a multistage negotiation paradigm for planning in a distributed environment with decentralized control and limited intercommunication (Conry, et al., 1986:1). The motivation for their work is the same as for this paper, in other words, to provide for the control of a complex communications system, without relying on a central planning function. They summarize the problem as follows:

- Goals are autonomously generated at nodes in the system
- The same system goal may be generated at more than one node, independently.
- Knowledge about local resource availability and potential goal interactions at each node differs from that at other nodes.
- Goal satisfaction in general requires nonlocal resources.

- The planning problem being addressed is, in general, overconstrained. A choice to satisfy some goals may preclude the satisfaction of others, so choice heuristics are necessary.

- Goals are prioritized, but this does not imply a total ordering with respect to priority (Conry, 1986:7).

The authors employ a form of negotiation, similar to contract bidding, to communicate primal needs. Using a carefully crafted arbitration process known as multistage negotiation, the "agents" responsible for proposing and communicating goals progressively learn how their demands will affect adjacent nodes (and ultimately the success or failure of the proposed goal) as other agents distribute information through the network. The messages that an agent receives fall into one of two categories: the first is a request to discover how other nodes will commit themselves; the other is feedback from nodes responding to a commitment request.

This form of information exchange enables nodes to determine locally if a goal is a good one or contains negative information reflecting non-local resource conflict (Conry, et al., 1986:10). The information exchange continues until no more changes take place, or alternatively, until a pre-determined time limit large enough to accommodate all expected transactions expires. In general, a node first propagates p-goals (primary goals) to neighbor nodes. In response they generate s-goals (secondary goals) to help out their neighbor. Should the neighbor not be able to help, the neighbor passes the conflict information on to others who in turn reconsider their s-goals. The negotiation-sequence is summarized below.

1. Each node examines its own p-goals, making a tentative commitment to the highest rated set of locally feasible plan fragments for p-goals (s-goals are not considered at this point because some other agent has corresponding p-goals).
2. Each node requests that other agents attempt to confirm a plan choice consistent with its commitment. Note that an agent need only communicate with agents who can provide input relevant to this tentative commitment.
3. A node examines its incoming queue for communications from other nodes. Requests for confirmation of other agents' tentative commitments are handled by adding the relevant s-goals to a set of active goals. Responses to this agent's own requests are incorporated in the local feasibility tree and used as additional knowledge in making revisions to its tentative commitment.

4. The set of active goals consists of all the local p-goals together with those s-goals that have been added (in step 3). The agent rates the alternatives associated with active goals based on their cost, any confirming evidence that the alternative is a good choice, any negative evidence in the form of nonlocal conflict information, and the importance of the goal (p-goal, s-goal, etc.). A revised tentative commitment is made to a highest rated set of locally consistent alternatives for active goals. In general, this may involve decisions to add plan fragments to the tentative commitment and to delete plan fragments from the old tentative commitment. Messages reflecting any changes in the tentative commitment and perceived conflicts with that commitment are transmitted to the appropriate agents.
5. The incoming message queue is examined again and activity proceeds as described above (from step 3). The process of aggregating knowledge about nonlocal conflicts continues until a node is aware of all conflicts in which its plan fragments are a contributing factor.

Remarks

Each of the preceding algorithms suggests ways to implement some aspect of a routing algorithm for a CNCE network. The Linear Programming, Maximum Flow, Minimum Cost Flow, and Shortest Path methods are important tools ultimately for system planning. All four optimize resources as their names suggest, making them useful for the initial layout of a network. However, in their present implementation each requires global knowledge of the system network and can not be directly incorporated into a distributed CNCE network without some form of modification.

The DNCS Algorithm 3 shows that by storing the current network status in each node of the network, the Shortest Path method can overcome the global knowledge limitation, but only at the expense of multiple system updates every time a node or link changes. Algorithm 1 exhibits the most promise as a feasible algorithm for CNCE network allocation. It meets the requirements of not needing global information yet goes about its business without a multitude of update messages. Using a two-sweep process, the first phase locates all connected nodes while the second phase returns to the sender information about the cost and route of the circuit. Although simple in concept and modular in design, Algorithm 1 relies on time-out messages to terminate. It also does not optimize, instead leaving a cost-ranked choice of possible routes for the technical controller to choose from. This open-endedness may not be much of a drawback as it allows the technical controller to deal flexibly with a wide range of constantly changing external information.

The constraint-based approaches provide useful insight on how to formulate a CNCE allocation algorithm. The Waltz algorithm in particular shares many similarities with the Minimum Cost Flow algorithm and Algorithm 1, yet it terminates with an optimal solution. Lastly, Multistage Negotiation suggests a way to minimize disruption among established circuit users during preemption, a problem of global dimensions that requires a distributive means of resolution.

IV. Design Implementation

This study employs a two-step approach to achieve resource allocation ends: incorporate the main features of a tactical communications network into an object oriented program (OOP), and graphically show how the objects react during search and allocation processes. More precisely, the following pages present the design implementation in terms of these core ideas: 1) design goals, 2) OOP classes, 3) saturation search algorithm, 4) commands and tools to facilitate route planning, and 5) graphics display. Although equipment constraints steered the design toward a Prototype Route Planner (PRP) to be used in the initial construction of a CNCE network, a related goal throughout remained the development of the Technical Controller's Assistant (TCA), a semi-automatic route planning system, distributed to every CNCE in a network, that would help technical controllers at these nodes deal with massive restructuring requests by finding viable routes for them. The two projects share common development stages; only the final applications differ. In the Technical Controller's Assistant, copies of a modified routing algorithm would be placed in each CNCE of a tactical network. When conditions warrant finding a new route, a human technical controller calls on TCA to find a route, and another operator authorizes the ensuing choice. In contrast, PRP simulates an entire CNCE network, permitting one person to build, modify, and generate circuits on a healthy or damaged network using the same principles as TCA.

Design Goals

This study derives it design goals from conversations with communications people who are familiar with CNCE networks. The following goal statements reflect constraints imposed by real-world conditions and perceived areas of improvement.

- The system should be able to pick a circuit path without relying on stored alternatives.

One problem with the current method of route planning is that technical controllers must rely on backup plans to restore a circuit, should it go bad. During peace-time this system works well, since malfunctions infrequently occur. The situation could be quite different during war, when coordinated attacks might render significant portions of the network inoperative. Circuit restoration under these circumstances will most likely proceed in an ad hoc manner as it is highly unlikely that a backup plan will cover the exact nature of the damage. As it stands today, technical controllers in surviving CNCEs will communicate with other controllers to determine the extent of the damage, then restore service to high-priority circuits, preempting low-priority circuits as necessary, relying on backup plans whenever they coincide with the actual damage.

- A restorative process should function without knowing the overall status of the network.

During the crucial first few moments of an attack, technical controllers throughout the network must learn of the network's global status before they can meaningfully restore circuits. This information-gathering period comes at a time when commanders need the damaged dedicated circuits most. Clearly an approach that bypasses or eliminates the fact-finding stage, would hasten restoration efforts.

- The system should provide path information but allow human intervention and authorization.

A fully automatic route-planning system, one that plots and activates a route autonomously, will most likely suffer from insufficient route-planning information, since some of the information needed to route a circuit could come from external sources. Currently, if a technical controller is aware of outside information, such as the location of an enemy attack, he can route circuits away from the battle. On the other hand, a fully automatic route-planning system

would pick an optimum route based on pre-programmed ideals, oblivious to the benefits a longer but safer route has to offer.

- The system must accommodate prioritized circuits.

Military communications networks differ from civilian networks in that the military preempts low priority circuits as necessary to ensure that high priority circuits get through. To be of benefit, a routing algorithm must reflect the same preemption decisions that a technical controller would make.

- The system should work with limited computer resources.

Each CNCE hosts a rugged but small computer that currently operates a database. The system has 64K of memory and permanent disk storage, but the computer has little room for additional functions and quite possibly may be inadequate for routing functions. The above goals sketch out desirable features for either the PRP or the TCA. The next section details how to quantify network features so as to be able to work towards these goals, from the perspective of the PRP.

Object Classes

There are two fundamental reasons for using object oriented programming as a tool to tackle the tactical communications route-planning problem. First, object oriented code extracts the salient features of a network, facilitating network models (this process is also known as abstraction, Booch, 1987:12). Whether capturing main ideas for a proof-of-concept prototype, or expanding those ideas into an operational system, an object oriented design groups data and

function into an object that behaves as would its real-world counterpart, thus permitting code-objects to accurately represent their physical counterparts.

Second, object oriented programming simplifies the code's design by breaking it into small, logical parts. By splitting the network into component classes (CIRCUIT, node, LINK, and CHANNEL), the programmer can detail the workings of each object. This modularity (Booch, 1987:12) yields simple objects with specific tasks, that when thrown together with other objects, work in concert to perform complicated procedures.

An unexpected benefit of using object oriented programming for this study is that simplifications made within the code suggest similar simplifications could be made within an actual CNCE network. The next chapter covers these simplifications and what they mean in more detail, the point being that it would be hard to develop and test improvements with hardware alone.

Before going further, a quick discussion on the choice of development languages is in order. Any language that contains object extensions can be used to model CNCE networks. Flavors, allied with the powerful Lisp language was used initially to carve out a rough draft of the main objects. The author's familiarity with Flavors, and an abundance of technical help paved the way for rapid conversion of ideas into code. However, after a week and a half of development, a decision was made to switch to a similar, but simpler language, PC Scheme. By changing to Scheme and its object oriented extension Scoops (Scheme Object Oriented Programming System), the author retained the lisp-like features of the first draft, and gained a development language that could execute on an small, IBM XT class computer (small when compared to the exotic Symbolics machine that Flavors resides on), (Texas Instruments, 1987:1-1). This theme of small computer development is in keeping with the design goals and realities of CNCE networks. Although Scheme cannot be used directly on a CNCE's computer, the fact that the entire development language fits in a small computer suggests that a final version could reside happily on a machine no larger than one with 640K of memory and a hard drive for permanent storage.

In an object oriented design, a good deal of time is spent analyzing how system sub-components interface, so that when it comes time to write the procedures that give objects character, a natural interactive hierarchy will already be in place. In PC Scheme, the class definition details each object's attributes. PRP uses five classes, IDENTITY, NODE, LINK, CHANNEL, and CIRCUIT, to represent physical entities and common properties. The following paragraphs explain the purpose and place of each.

IDENTITY Class. A class acts like a template, defining features common to all its offspring. These offspring, referred to as instantiations, share the same instance variable names defined by their class, but the contents of these variables vary depending on the instance. For example, all objects in PRP share a class called IDENTITY. The IDENTITY class owns two instance

```
(define-class IDENTITY
        (instvars Name
                  (Status 'Alive))
        (options (gettable-variables Name)
                 (settable-variables Status)
                 (inittable-variables Name)))
```

Figure 4. Identity Class Declaration

variables, NAME and STATUS (figure 4). NAME stores the title given an instance while STATUS stores its condition (dead or alive). No two objects share the same name, yet any object will respond with its given title whenever queried through the common instance variable NAME. The

same holds for STATUS. Some objects may be dead, and others alive; querying an object's STATUS determines the answer. Unique among classes, the IDENTITY class does not represent a physical entity in the tactical network. Instead, other classes inherit this basic information, reducing the complexity of the code.

NODE Class. A node represents a CNCE. It contains procedures that implement the routing algorithm, serves as the entry point for circuit-requests, and the exit point for circuit-replies. Nodes connect only to links (trunks) and store a finite number of circuit-request and circuit-relay messages that pass through the node enroute to other nodes. An instance variable called REQUESTS (figure 5) stores the request-messages in list form. Likewise, an instance

```
(define-class NODE
        (classvars (Population 0))
        (mixins IDENTITY)
        (instvars (Rectangle (make-window #F #T))
              (Serial-Number (set! Population (1 +
Population)))
              Requests
              Replies
              Links)
```

Figure 5. Node Class Declaration

variable called REPLIES retains final path information, also in list form. Since a node talks only to those links attached to it, it must request channel and circuit information directly from the link.

Note that a node is "unaware" of its node neighbors and the channels that carry message traffic for the link. In an analogous sense, a technical controller deals primarily with the trunk lines that terminate at his CNCE, using indirect trouble-shooting methods to obtain channel and neighbor CNCE information.

LINK Class. A link represents a trunk line and functions so as to propagate information from end node to end node. A link knows which nodes mark its boundaries and it knows the channels assigned to it, just a microwave trunk line can be characterized by its termination points and the twenty-four T-1 channels it carries. This information is stored in list format in the NODES and CHANNELS instance variables (figure 6). The link however, does not know if a channel has been assigned to a circuit. This information must be found out by querying the channel and

```
(define-class LINK
        (mixins IDENTITY)
        (instvars (Score-Board (make-window #F #F))
                Nodes
                Channels)
        (options gettable-variables
              settable-variables
              (inittable-variables Nodes)))
```

Figure 6. Link Class Declaration

parallels the signal check a technical controller makes to determine if a channel carries data. The query process breaks from reality in that a trunk can not interrogate a channel. Instead, the technical controller determines the information using test signals or representative information

stored in a data base or written on cards. In any event a link, like its trunk counterpart, effectively bundles all channel information into one entity for easier handling.

CHANNEL Class. Channels represent a resource capable of carrying messages. Channels know which link they belong to and the circuit they carry. This information resides in LINK and CIRCUIT instance variables respectively (figure 7). A real-life channel can be tested for a carrier signal to determine if it carries a circuit, while the absence of a carrier signifies that the channel remains unused (or has malfunctioned). In actuality a channel does not associate itself

```
(define-class CHANNEL
        (mixins IDENTITY)
        (instvars Link
                (Circuit (active 'Spare #F
auto-connect-Circuit)))
        (options gettable-variables
```

Figure 7. Channel Class Declaration

with a link except by physical location and it does not know the identity of the circuit it carries, but records maintained at a CNCE will show channel-circuit pairs and the trunk to which a channel belongs. For this reason the CHANNEL class permits direct communication with links and circuits.

CIRCUIT Class. Circuits constitute the end result of the Prototype Route Planner. They carry messages from one destination to another via intermediate points as do their physical counterparts. This object includes a priority attribute and a list of channels that form a path

between the two end points. This representation departs from reality in that a real circuit does not keep track of the channels it uses. Instead, a data base or card system describes the allocated channels. The circuit is assigned a priority but again, it does not intrinsically know that priority while the data base does. This poses a real world problem because a circuit's priority can change while not immediately affecting the resources allocated to it. Later preemptions can cause problems if the "robbing" circuit is not aware of the upgraded priority of the "victim" circuit.

Message Implementation

Messages govern network operation in this implementation, and although they play a major part in a CNCE network, they do not warrant a separate class for two reasons. First, an object-like message representation, however suitable for computer simulation, could not be applied directly to operational use due to the inherent complexity of the message's data structure (objects cannot be passed between computers). Instead, a list format translates easier due to its string-like appearance. Second, let us assume that a message can take object form. Then for every message created, additional copies would be needed for distribution throughout the network. As these copies move about, they paradoxically store path information that distinguishes them from each other. Unfortunately, once instantiated, an object can not be split, and even if it could, one would run into a data structure problem – object oriented languages all have convenient ways to create objects, but few include a graceful way to destroy objects without contaminating related data structures. For this reason, object instantiations under this format would continue to grow until they exhaust computer memory, or until program termination when the computer, upon releasing memory, effectively eradicates all objects.

Saturation Search Algorithm

For several reasons the saturation search algorithm first introduced by Ludwig and Roy fulfills the requirements for a route planning system: 1) The saturation search algorithm provides a convenient way to work around damaged nodes and links without devoting large amounts of

computer memory to monitoring the network's operational status; 2) The route pruning features inherent in the algorithm reduce the number of messages that circulate through the network; 3) The algorithm does not rely on global information, facilitating its use in a distributive environment. Both the Prototype Route Planner and the Technical Controllers Assistant benefit from this algorithm and the prioritization scheme proposed by Computer Services Corporation for use in the DPAS Network Control System.

Figure 8 shows the message format used in the PRP. As the message is really a list, it can be easily copied and modified using Scheme operators. In PRP it remains in list form as it moves from link to node. In the TCA, the message would be converted to a string, then broadcast over trunks to adjacent nodes for further use. Three messages, request, reply, and reject, dictate the algorithm's mode for nodes (figure 9) and links (figure 10). A request-message, generated by a technical controller through the use of the Make-Circuit command, initiates the algorithm. The origin node builds the request-message, repeating it to every link it connects to. Each link passes the message to the node at the opposite end. The node records the message then relays it to every connecting link. As long as a single link connects an isolated node to the network's bulk, a request-message sent by the node will propagate through the network reaching every other node in the net. This tenacious quality proves beneficial for tactical networks as it inherently protects the network from CNCE and trunk damage. Redundant message traffic is the price paid for survival.

Message Attenuation. If not for certain attenuating principles, request messages would echo forever within a network. One such attenuator, message cost, works even in a prioritized system. Consider a request for an A-priority circuit. Its request message shows [1 0 0 0] as its priority vector. Next, assume that one of the paths traveled by the request-message yields one B-priority channel, one C-priority channel, and one spare channel as a possible path combination. This three-hop path shows a cumulative cost of [0 1 1 1], meaning one B-priority circuit, and one C-priority circuit would have to be sacrificed, and one spare would have to be expended to complete the A-priority request. If at this same node, a subsequent request-message discloses a

cumulative cost vector of [0 1 2 0] , the node would not transmit this message to its connected links as its proposed path preempts an additional C-priority circuit, making this route more

(Circuit-ID Mode Destination Priority Previous-Cost (Path))

where:

Circuit-ID = Circuit-X

Mode = Request, Reply, or Reject

Destination = Circuit termination CNCE

Priority = [1 0 0 0], [0 1 0 0], or [0 0 1 0], for A, B, or C priorities respectively

Previous-Cost = [? ? ? ?]

(Path) = (Destination, Link-Y, Node-Z...Origin)

Figure 8. Message Format

expensive than the first. The algorithm goes one step further by blocking alternate routes of identical costs, so that only inexpensive routes generate repeat messages.

A second message attenuator works on an allocation principle: if a request message warrants preemption of existing circuits, relay the message, otherwise terminate it. This principle weeds out saturated path segments in favor of unsaturated paths. For example, if an A-priority circuit-request reaches a node connected to two links, each with single channels already dedicated to circuits, the following would happen. The node asks the first link what priority traffic it carries. When it replies B-priority, the node relays the message since an A-priority request can

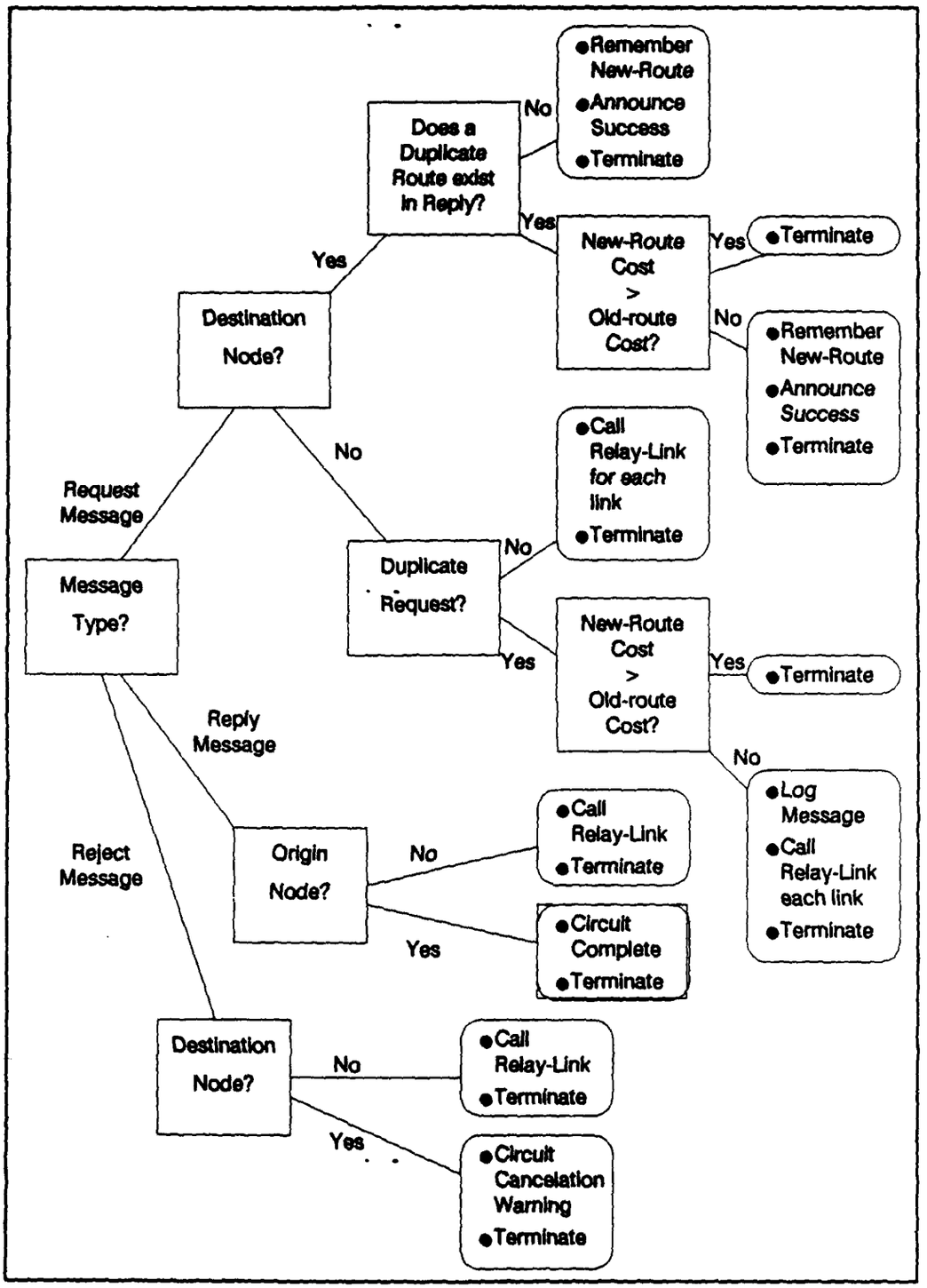

Figure. 9 Node Flow Chart

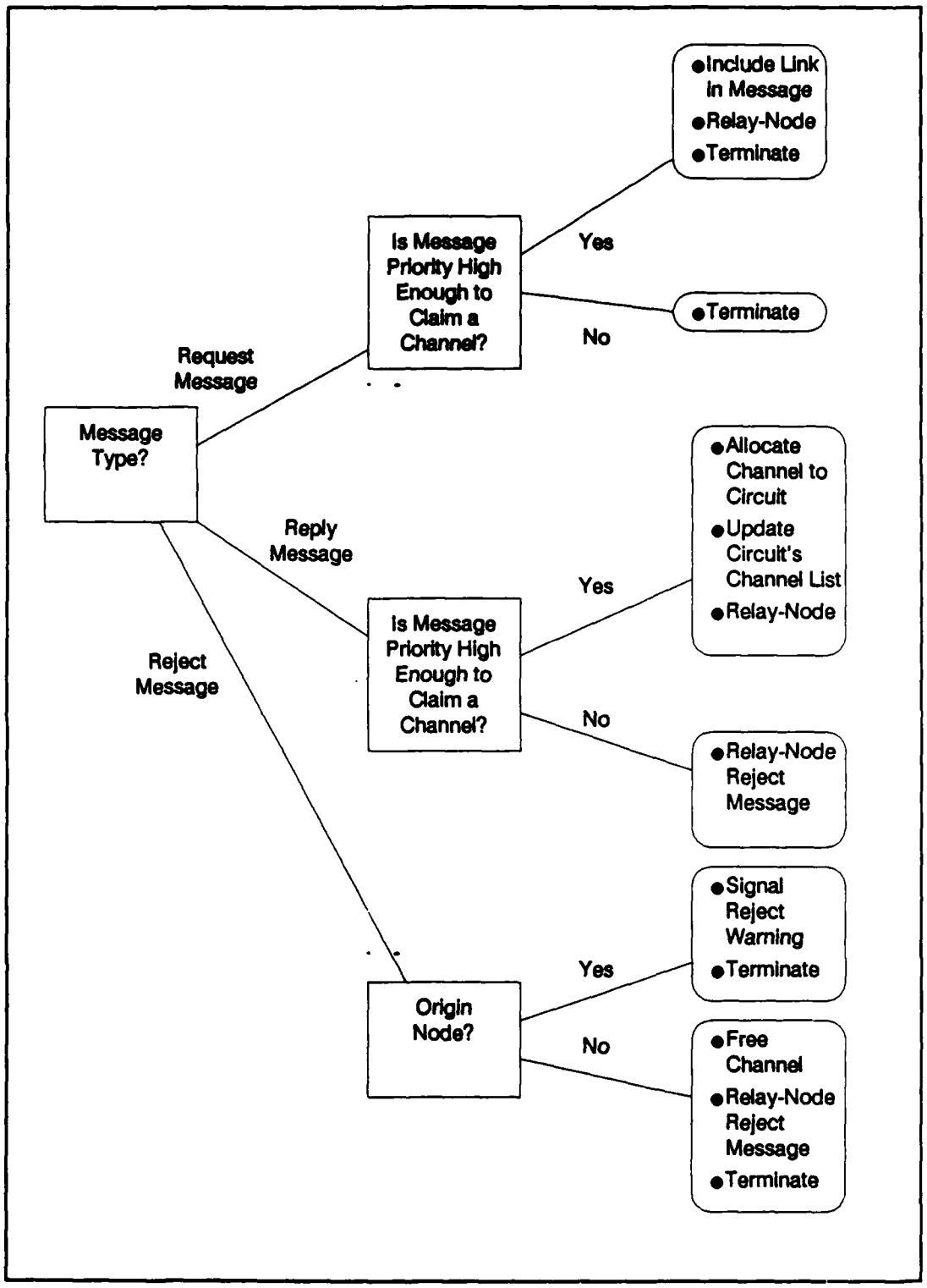

Figure. 10 Link Flow Chart

preempt B and C-priority channels. When the second link answers that it carries another A-priority channel, the node bypasses this link since a request can not preempt channels of equal or higher priority. During this channel check, no channels are allocated. To do so would be premature since a request message has no guarantee that the proposed path it carries will work, or that the destination node even exists.

Message Propagation. As a request message propagates through the network, each node stores a copy of it in the instance variable REQUESTS. REQUESTS always stores the initial request message, updating the message only if it learns of a less expensive route. To prevent confusion, each circuit-request must have a unique identity to distinguish it from circuit-requests generated elsewhere in the network. In little time a node's REQUESTS depository would fill with new and dated request-messages, posing a memory capacity problem if it were not for the fact that a first-in first-out (FIFO) buffer limits the number of messages. The size of the buffer determines the number of simultaneous circuit-requests the network is capable of handling. If too large, the algorithm could suffer performance penalties due to excessive search times. If too small, the buffer will dump path information forcing a node to pass on a message it might normally terminate.

Given a viable network and a circuit request, the request message will reach its destination. The destination node places the path candidate from the message in the instance variable, Replies. As other alternate route messages reach their destination, they too contribute paths to Replies providing the path cost does not exceed that of the least expensive route stored so far. Thus, unlike a support node that passes only least-cost information, a destination node retains low and equal cost alternatives so that the technical controller can choose the best route based on external information. This feature works toward the design goal of incorporating outside information into the choice of routes.

Once a request message's path can be determined, the algorithm then must allocate channels along the path and in doing so establish the circuit. This allocation phase works from the

destination node backwards to the origin node and commences when the technical controller specifies a route with the Pick Route (PR) command. The destination node first records the choice in Replies and then generates a reply-message for channel allocation. Unlike the original version of the algorithm which relies on route information stored in Requests to back-trace the route, this version carries path information with it, precluding any loss of path information should the Requests buffer flush the circuit's request message.

Upon receipt of a reply message, each link along the route picks a channel to allocate to the circuit. The same identification process occurred earlier when each link verified it could support the circuit. A second check, warranted by virtue of the fact that a competing circuit may have stolen the previously identified channel, reaffirms that a channel still exists and confirms the circuit's viability to this point. As long as a suitable channel exists, the allocation process continues. If not, the node where the breach occurred generates a reject message, signalling that the allocation process can go no further.

The reject mode de-allocates all channels recently assigned to the circuit and warns the technical controller who picked the route to choose a new one from the alternates. If no alternates exist, the technical controller who originated the request will re-initiate the circuit after waiting a specified period of time for the circuit to go through. The rejection process frees channels by undoing each allocation one at a time, from the point of conflict to the destination node. When the reject message reaches the destination node, it warns the technical controller via a text message or audio alarm.

Command Language

Four commands, Make-Circuit (MC), Pick-Route (PR), Dead, and Alive, make up the vocabulary in the Prototype Route Planner's language. The first two commands interface directly with the route planner, while the last two control external changes to the network. In a TCA version, the technical controller at a CNCE would use a simplified form of MC and PR to build

network circuits. The Dead and Alive commands would not exist as the environment assumes responsibility for incapacitating equipment.

Make-Circuit Command. From a TCA perspective, the Make-Circuit command lets a technical controller express the need for a circuit from his CNCE to another. Either end can invoke the command, but a protocol that eliminates the ambiguity would also preclude redundant circuit requests. The author suggests that the originating CNCE should be subordinate to the destination CNCE since the destination CNCE ultimately authorizes the route choice (but only if the junior node identifies the need).

In the PRP implementation, the network planner enjoys a world view of the network and assumes the technical controller role for every node, requesting circuits between any two CNCEs without the need for a protocol. The command takes the form: (MC Origin Destination Priority) where MC denotes the Make-Circuit command, Origin determines the node that will generate the request-message, Destination identifies the terminating node, and Priority takes A, B, or C as priority values for the proposed circuit.

The origin node identified by Make-Circuit creates a serial number for the tentative circuit to distinguish it from other requests, generates a request message, then hands control of the request to the network for execution. The Make-Circuit command goes no further. It is up to the algorithm, the destination controller, and fate to determine if the circuit will go through. Of course the route planner will have a good idea if a circuit will go through since he determines the environment and monitors messages as they course through the network. This does not mean however, that Make-Circuit is a one-shot forget-all command. In the TCA, if the originator does not receive confirmation of the new circuit's existence after a sufficient period of time, then Make-Circuit should be tried again; the delay may be due to channel conflicts arising in ways alluded to earlier.

Pick-Route Command. The Pick-Route command authorizes one route for channel allocation and sets in motion the circuit building process. In a typical situation, a route planner

uses Pick-Route after all message passing activity stops and time permits the finalization of circuit. The command takes the form: (PR Number) where PR denotes the Pick-Route command, and Number determines which node should be examined for unresolved circuit paths. Pick-Route need not be applied immediately after a Make-Circuit request since Replies stores a finite number of solutions in a first-in first-out buffer just as Requests does. But if the route planner fails to act in time, a full buffer will flush unresolved circuits as additional circuits come in.

The Pick-Route command displays circuit information in tabular form, on a first arrived, first answered basis. Pick-Route numbers the choices, displays path cost, shows the routes, then prompts the planner for a choice. The number one choice will always show a least expensive route. Successive choices will show routes of equal or greater expense. Thus a route planner can build a low cost network by picking routes at the top of the list. The planner however, has the freedom to pick any route or abort the selection process. If a legitimate choice is made, Pick-Route constructs a reply message which is dispatched to the destination to complete the allocation process. The command also creates an instance of the circuit, named after its circuit ID to accommodate channel information incurred while building the circuit. This instantiation creates the circuit as an object even though it owns no channels at that moment.

Pick-Route also codes the selected route information within Replies and preserves the unchosen alternates. If the chosen route does not go through, the ensuing reject message alerts the route planner to re-apply Pick-Route to choose an alternate route. When used in this manner Pick-Route annotates the ill-fated circuit so that it will not be tried again by mistake

Dead Command. The Dead command kills any instance of a node, link, channel, or circuit class. The command works in conjunction with the IDENTITY class which is an inherited part of all objects. If an object is killed with the command (Dead Object), the status instance variable of that object will be set to 'dead. A careful examination of the methods (procedural code) assigned to each class of object reveals that an object will not perform programmed tasks if it is dead. This is the result of an if-clause or other related conditional statement at the beginning of

each method. A dead object returns nil as its value, adding no information to the network and demanding no work from associate objects. In summary, a dead object still takes up space in the network, but does not respond to inputs.

Alive Command. The Alive command rejuvenates any dead object, returning its Status instance variable to the value 'alive. The conditional statements at the beginning of each method check for an alive object before they continue execution. An object can be killed and reincarnated as many times as necessary by the route planner, but the user is warned that PRP makes no effort to remove dead objects. They reside in memory until the session terminates and in large network simulations, could consume a good amount of memory.

Inspection Tools

Inspection tools fall into two catagories, those that help the route planner, and those that help the programmer. The first category includes: Channels?, Circuits?, Cost?, Links?, Nodes?, Living?, and Look, which enable the route planner to determine associations between network objects. The second category, Replies? and Requests? are used to monitor the messages stored at a node and for this reason offer little useful information to the route planner, but do help the programmer make sense out of a long list of data.

The route-planner inspection tools use the natural hierarchy imposed by the object classes to obtain information. The chain-like association starts at one end with the NODE class. Nodes recognize links since the LINKS instance variable resides within the NODE class. Likewise links recognize nodes and channels; channels recognize links and circuits; and circuits, at the other end of the chain, recognize channels. The hierarchy naturally limits how the objects interact, but this programming convenience proves difficult when it comes to extracting information from non-adjacent objects as they do not associate with each other. For this reason, the inspection tools offer an easy way to extrapolate information outside the chain. For example, the query (Links? Node-1) causes Node-1 to output a list of the links it owns (the list will not be readable due to the internal representation of objects, but the Look command presented later overcomes this

restriction). Since Node-1 need only send the contents of its LINKS instance-variable, one would not expect this task to be too difficult – and it's not if the information can be requested directly. However, how can one ask Node-1 about the circuits it owns when it does not have a CIRCUITS instance-variable? The answer lies in the hierarchy. Node-1 asks all the links connected to it what circuits they carry. The links in turn pass the question to their respective channels. The channels return the answer to the links and the links pass the information to Node-1 which collects the results and offers it as if it directly owned channels. This convention works for all classes of objects, as shown in Figure 11.

Of the remaining commands, Look is used to reveal the given name of an object while Living? asks if an object is dead or alive. Look is used in conjunction with the above object query tools in the form (Look (X? Object)). Look substitutes a name for the computer's coded representation and will work for any object as long as it possesses the Name instance variable inherited by the IDENTITY class. The query (Living? Object) returns a simple "dead" or "alive" response for the object in question. The tool uses information from the Status instance variable to determine the nature of an object. One should note that while Living? elicits a reply regardless of death, other tools will not work the same way. If an object is dead, it will not be revealed by Look or any of the object-query tools since the object will not respond to questions. Thus only participating objects appear as answers to probes by the inspection tools.

Graphical Representation

The Prototype Route Planner (PRP) displays individual nodes and links, and summary channel information so that the route planner can observe the effects that circuit requests have on healthy and damaged networks. The PRP uses rectangles arranged in a ring to represent CNCEs. The circular arrangement, although it does not preserve true network geometry, does retain the one-to-one functional relationship between links and nodes. The open area in the middle of the ring provides space for pop-up windows like the one that displays proposed route information for

the Pick-Route command. This arrangement also minimizes the number of nodes that must be re-drawn, since pop-up windows rarely overlap the nodes. As a benefit, the lines (trunks) that

Objects / Inspection Tools	Node	Link	Channel	Circuit
Nodes?	If alive, returns itself as an object.	Returns the nodes, if alive, at either link end.	Returns the nodes, if alive, at either channel end.	Returns all living nodes the circuit passes through.
Links?	Returns all living links owned by a node.	If alive, returns itself as an object.	Returns the link, if alive, that carries the channel.	Returns all living links the circuit passes through.
Channels?	Returns all living channels owned by a node.	Returns all living channels owned by a link.	If alive, returns itself as an object.	Returns all living channels the circuit passes through.
Circuits?	Returns all active circuits that pass through a node.	Returns all living circuits owned by a link.	Returns the circuit, if allocated, or spare if not.	If alive, returns itself as an object.

Figure 11. Multi-Object Inspection Tools

connect one CNCE to another do not overlap due to the ring's symmetry. Next to each line a four-place vector displays how the channels have been allocated. The first number details the number of channels used to support A-priority circuits; the second, B-priority circuits; the third, C-priority circuits; and the fourth, spare channels.

The Prototype Route Planner highlights individual nodes and links as they participate in the allocation algorithm. Dead objects show up in a ghost-like gray color, while nodes and links take purple and green hues respectively. During the algorithm's final channel allocation phase, the links that support the circuit path turn red to temporarily identify the new circuit's route. In

addition, as the algorithm commits channels to the circuit, each link updates the allocation vector to show its channel content.

V. Conclusions and Recommendations

This section summarizes key concepts developed in the first four chapters and then examines the study's results. More specifically, it assesses the advantages and limitations of the Prototype Route Planner, the saturation routing algorithm, and predicts how the algorithm would act if embedded in a tactical network in the form of a Technical Controller's Assistant. Conclusions follow and recommendations for future work completes the section.

Summary

Technical controllers who run CNCE networks face time-consuming resource allocation problems when they configure or maintain a network. The process of matching equipment to need takes months of planning, and continues in the field, where technical controllers manually accommodate last minute changes. To maintain a deployed network, controllers at each CNCE generally swap spare-circuit resources for failed dedicated circuits according to a pre-determined backup plan; however, if massive failures occur as they could during an attack, the controllers would preserve high priority circuits at the expense of spare and low priority circuits. But before they could make effective restorations, the controllers as a whole would have to learn the extent of the damage – a confusing, lengthy process at best that could be curtailed if there were some means to automatically allocate resources.

An undirected graph describes the node, link, and channel objects in a CNCE network, abstractions that assist an object oriented design for the allocation problem. A CNCE can be represented as a node, a trunk line as a link, and channels in the trunk as a multi-element vector that counts the number of channels assigned to high, intermediate, and low priorities, and spare status. Each circuit is assigned a priority that passes to the channels that support it in the network. During the allocation process, a high priority circuit can seize channels from circuits of lesser priority only. Regardless of a circuit's priority, technical controllers try to allocate spare channels

first, and high priority channels last. It follows then that the cost of a completed circuit is the sum of the channels used, by priority. Although backup plans detail how to compensate for some losses, massive outages force ad hoc solutions that could be avoided if a systematic method of resource allocation were applied.

Operations research search techniques, communications protocols, and constraint-based sources were consulted in an attempt to find an algorithm that can allocate channels in a damaged network. Of the traditional routing algorithms – linear programming, the max flow algorithm, the minimum cost flow algorithm, and the Dijkstra shortest path algorithm – all require absolute knowledge of a network's status before they can function and thus were not suitable during massive outage situations. Of the DPAS network control algorithms, both the saturation search algorithm and a modified shortest path algorithm work in uncertain environments, but the saturation routing algorithm was chosen for implementation due to its simplicity, and low message traffic. Finally, although the Waltz algorithm and multi-stage negotiation methods suggest that they would work in an CNCE network, neither appeared better than the saturation routing algorithm for object oriented programming implementation.

Having identified network objects and a routing algorithm, the Prototype Route Planner (PRP) was deemed the best way to demonstrate resource allocation in a tactical network since a fully functional Technical Controller's Assistant (TCA) was not physically practical. PRP, which resides in a single personal computer, simulates a healthy or damaged network and employs the saturation routing algorithm to help a technical controller find viable routes under various conditions. A TCA demonstration on the other hand, would have shown that a hard-wired network with concurrently running copies of TCA in every personal computer node, could find viable circuit routes among the surviving links and nodes of a disrupted network.

The common design goals call for a system that finds routes without resorting to prior planning, that functions without knowing the overall network status, that permits human

intervention, that accommodates prioritized circuits, and that works with limited computer resources.

PRP's object classes include NODE, LINK, CHANNEL, and CIRCUIT classes and a subordinate IDENTITY class that standardizes name and status information. The objects form a hierarchical ladder where nodes talk only to links, links talk to nodes and channels, channels talk to links and circuits, and circuits talk only to channels. The saturation search algorithm controls how nodes and links react to requests for new circuits. A request message ripples throughout the network starting at the node where it is inserted, travels along all links that connect that node to its neighbors, and so forth until every node hears the message. If connected by a usable path, the destination node will eventually receive the message, store the route's cost, and signal the technical controller for authorization. Each node rejects duplicate messages when they equal or exceed the cost of the previous message and repeats subsequent low cost messages thereby propagating a lowest cost solution. The technical controller at the destination picks a route, which starts a reply message back along the winning path. On the way channels are allocated to the circuit or if for some reason none exist, the last participating node in the allocation chain generates a reject message that de-allocates all previous channels and warns the technical controller to choose an alternate.

PRP uses commands to kill and revive network objects and to manipulate the network. Inspection tools permit the user to assess how individual objects relate to one another. Finally, PRP uses graphics to depict the allocation process.

Assessment

The algorithm used in the PRP behaves differently than the same algorithm would for a Technical Controller's Assistant due to the single-computer implementation of PRP. Although PRP network objects appear as separate entities, the computer processes only one object at a time.

Thus, a command that seems to manipulate several objects at the same time, really operates on the objects sequentially. As a result, PRP does not enjoy the speed that TCA would with parallel-path processing capabilites.

The amount of time that PRP requires to execute its search algorithm depends on the number of nodes and links in the network. If nodes outnumber links (possible only in the rare case of a linear node arrangement), then the number of nodes that handle the message could decide how quickly the algorithm finds the solution. On the other hand, if there are more links than nodes, then the time spent exploring each link determines the time to complete the algorithm. As the routing algorithm in PRP behaves like a depth-first search (DFS), one would expect a search time similar to DFS's O(max (n,e)), which says that DFS executes in time based on the maximum number of nodes or edges (links) in a network (Aho and others, 1974:178).

PRP requires at least the same amount of time to function, but a closer look reveals that the number of channels in a link determines how fast the implemented algorithm will run. During request message propagation, each link polls its channels to determine if it can accommodate the circuit request. As the implemented algorithm executes serially, it can not advance until all the channels in the link have replied. Thus PRP executes in time proportional to the number of channels in the network and consumes memory space based on the number of channels since each channel use a finite amount of memory.

In the TCA version of the saturation routing algorithm, the network nodes parallel process request messages assisted by the links which determine channel-status non-sequentially. No longer bound by the constraints of depth first search, the parallel processing of the search messages diminishes link-quantity as a determinant of execution speed. Furthermore, if channel status information can be gathered and consolidated independent of request message use, then channel quantity no longer dominates the algorithm's execution time. Thus the search algorithm in TCA runs in time O(n), or according to the number of nodes.

Conclusions

PRP represents an initial attempt to route circuits in a tactical network with an algorithm that takes into account real-world constraints. The fact that this can be done on a limited scale suggests that an inexpensive route planner could be developed for actual use. Even more so, an inexpensive retro-fit could be designed for existing CNCEs, offering circuit routing flexibility and freedom from contingency plans as a by-product.

Although designed for parallel operation, the same basic tenets of message passing, cost determination, and channel allocation apply equally well to PRP's sequential execution of the algorithm. Thus, even though PRP explores nodes in an order different than TCA, PRP will produce a shortest path solution as would TCA. If all the route planner needs is a few good routes to choose from, then PRP will provide them, albeit slower than would a true parallel design, but with the ability exclude damaged components. From a time perspective, PRP searches a small network of four nodes, five links, and twenty channels within ten seconds. This time will increase as the number of nodes, links, and channels in particular, increase.

Channel priority encoding, which is pivotal to the design of PRP, would play an even bigger part in a TCA implementation. It also requires modifications to support the algorithm. Consider a request message that enters a tactical network supported by TCA. The origin node propagates copies of the message across every connected link, but before doing so it examines the capability of the link's channels to support the message's priority. Channel-status buffers at either end readily provide this information as they accumulate the priority attributes periodically, channel by channel, by polling the channel directly. The key is to make priority information available by hiding it in the message or the carrier so that the same information appears throughout the network. A broken circuit can not pass the priority code to the channels allocated to it so those channels revert to spare status for later circuit regeneration. The same technique permits rapid cost assessment for individual channels and larger links.

A CNCE network would be more tolerant to damage and easier to maintain if it incorporates a modified version of the saturation search algorithm. By encoding priority information within the active circuits that ride the network and extracting it at each CNCE, a computer at each node could send, pass, and receive routing messages that work in aggregate to find successful circuit routes without prior knowledge of the network's condition. A tactical network that incorporates this algorithm requires no pre-planned back-up routes since the algorithm determines the best route based on existing circuits and working equipment, while supplying the same information that technical controllers use now to maintain circuits.

Recommendations for Future Work

Although the PRP shows that a search algorithm can be adapted for use in a tactical communications network, more work is needed to accurately model CNCE network information. Several possible options come to mind: 1) Refine PRP object components so that they accurately embody the detail (bit rates, priority codes, etc.) needed to layout an actual tactical network. 2) Translate the algorithm's object implementation into Flavors for incorporation into GUS, an existing network mapping program written by graduate students at Clarkson College that runs on a Symbolics work station. 3) Write a mouse-based network drawing system like GUS that "hooks" into Scheme and converts mouse inputs into a network description like the network files included in appendix B. As an alternative, one could rewrite the network objects in C++ or Ada to take advantage of existing graphics routines.

PRP could profit by a system that stores the results of a networking session for later continued analysis. A feature like this stores the machine's environment in a file for storage, easing later recall of the network's transient state. Scheme supports this concept, but PRP does not implement network-capture at this time.

One final suggestion for further research lies in the area of network protocols. If circuit-ID and priority information can be incorporated into the header of a communications protocol, then

CNCEs could use this information to plan routes the same as PRP. The challenge, is to find a way to insert this information into existing networks.

Appendix A: Prototype Route Planner Operating Instructions

The Prototype Route Planner (PRP) helps you find routes for prioritized circuits in a tactical network. This manual tells you about the requirements and features of PRP and shows you how to use the system to experiment with networks of your own. The first portion of the manual (System Requirements, Creating Networks, Installation and Start-Up) discusses preparatory instructions, while the last half (Commands and Inspection Tools), covers features used while running PRP.

System Requirements

The Prototype Route Planner (PRP) requires an IBM compatible computer host, and PC Scheme version 3.0 to interpret commands and draw diagrams. Furthermore, PRP needs an EGA color graphics display and works best on computers with a Winchester (hard) drive. If you have the above configuration,(i.e. Zenith 248 with the EGA color monitor) the program will work as intended.

Creating Networks

Although you can create specific instances of nodes, links,and channels via keyboard commands during a PRP session, a network's components should be defined prior to using PRP.Any word processor that writes to an ASCII file, including Scheme's text editor, will do the job. First, retrieve one of the practice networks provided with PRP, then use the word processor to modify the nodes,links, and channels, so that they reflect the features you want them to have. If you need to make additional objects, just copy a similar object and update its name and instance variables with the necessary values. As you create objects that connect to each other, be aware that connections are bi-lateral; that is, a node has record of the links it can talk to, and each link completes the connection by including that node in its instance variable NODES. When you have

finished making and connecting the network's components, be sure to save your network under a new name so as not to write over the practice network.

Installation

This section tells you how to prepare your computer to run PRP. If you have already installed PC Scheme according to the following instructions, go to the Start-Up section, otherwise install PC Scheme by placing the distribution disk in the A: drive, then follow the installation directions in chapter 2 of the TI Scheme User's Guide. If you have an IBM compatible computer with EGA color monitor and Winchester disk drive, the installation command will look like:

A:INSTALL C: \SCHEME W

where \SCHEME is a directory created by the installation program that contains all Scheme executable code, and W indicates that the computer has a Winchester drive. If your system has only floppy disk drives, refer to chapter 2 of the User's Guide for installation details. Upon completion, you will be in the C:\SCHEME directory.

Start-Up

This section tells you about the batch command that starts PRP. Before using it you must know that it changes the computer's path to read from the hard and floppy disk drive. In this configuration, all PC Scheme code is accessed from the C: drive while all PRP code is accessed from the A: drive. After setting the path, the batch command then calls PC Scheme, and automatically loads PRP for PC Scheme to interpret. To start PRP make A: your current directory and type:

PRP

The initialization process takes about 12 seconds on an AT class machine. When it completes, you will see a blank screen except for the bottom line which contains the command prompt [1] , followed by the cursor and the status line. All commands entered from the keyboard will be echoed at the prompt. Also, PRP returns messages to you via the prompt. As an alternate means

of installation, you can copy the contents of the PRP disk to its own directory on the C: drive and change the path in PRP.BAT so that the computer can find this new directory. This alternate configuration permits faster code loading since all code is read from the Winchester drive. Additionally, it frees the A: drive for external use. However, since PC Scheme retains all loaded code in memory and does not free memory by writing to disk, this alternate installation method will not improve execution time.

Loading a Network

This section tells you how to load a network into PRP, since PRP contains no network information when you first start it. You could build a network one object at a time by defining them with Scheme's make-instance command, but it is much easier to load a pre-defined network. PRP comes with several practice networks that can be modified as desired. For example, to load the practice network STICK type the command:

(LOAD "STICK.FSL")

As this is a Scheme operation, parens bracket the word load, and quotes denote the file name. In the above example, the command will load the Stick network which is stored in a fast-load or .fsl file format. Fast-load files are text files that have been processed for quicker loading. PRP can also load source files (.s files), but these files take longer to read. Any pre-defined networks you create start out as source files (and should have an S extension). These files can later be converted to object and fastload files at your discretion. However, unless you discover a network worthy of repeated use, source files are the easiest to read and manipulate. The word OK will appear at the prompt if the file loads normally. If instead you see the prompt [Inspect], then PC Scheme has found an error. Since the PRP command line is only one line long, you will not be able to read the diagnostic message. Type

(NORMAL)

to change PRP to text mode, and try to load the file again. This time you will be able to read the diagnostic message. Correct any errors you find and return PRP to graphics mode by entering

(CLEAR)

before loading the network file. In general, use this screen toggling technique any time you bump into operating errors.

Commands

Four commands, Make-Circuit, Pick-Route, Dead and Alive control network objects.

Make-Circuit. The Make-Circuit (MC) command starts the search for a path from the initiating node to the destination node. The command includes the circuit's priority in the form of a single letter, A through C, and the starting and terminating nodes, referred to by number. For example,

(MC 1 2 'A)

requests an A priority circuit from node 1 to node 2. Note that the A is preceded by an apostrophe.

Pick-Route. As PRP searches for routes, it occasionally beeps to let you know that it found a good path. That is your cue to use the Pick-Route (PR) command on the destination node when the search activity stops. The Pick-Route command lets you act as the technical controller of a CNCE. For example, if you want to authorize a route at node 2 type

(PR 2)

PRP will display all reported routes for the first circuit in node-2's buffer. The display will show cost and route information and it will ask you to select one or none of the choices. To authorize one of the routes, simply enter its corresponding number and press the return key. When you do, PRP will generate a reply message that works its way back along the route, allocating channels to the circuit as it goes. If PRP can not find a suitable channel during this allocation process, it will warn you and ask that you pick an alternate route. If there is no alternate, the circuit can not be complete, and the request should be re-initiated from the originating node. If there is more than

one circuit in the node's buffer, PRP will display the second set of routes after dispatching the first. This process continues until the buffer is empty. If you mistakenly use the Pick-Route command on a non- destination node, PRP simply replies that there are no more routes.

Dead. The Dead command kills an object or a list of objects. It does this by changing the object's STATUS instance variable from 'ALIVE to 'DEAD. Dead objects, drawn in gray, do not respond to inspection questions and do not participate in the route finding process. For example, to kill NODE-1 type:

(DEAD CNCE-1)

Multiple objects can be killed as in the following example:

(DEAD (LIST CNCE-1 LINK-1 LINK-3 CHANNEL-12 CIRCUIT-9))

Alive. The Alive command rejuvenates dead objects by changing their STATUS instance variable from 'DEAD to 'ALIVE. Living objects show up in color and respond when queried. The Alive command revives single or multiple objects using the same syntax as the Dead command above.

Inspection Tools

The inspection tools Nodes?, Links?, Channels?, Circuits?, Requests?, Replies?, and Look, permit you to ask any object about its relationship with other objects. In other words, you can ask an object about the nodes, links, channels, and circuits that it owns and you can directly inspect the REQUESTS and REPLIES instance variables in a node. All inspection tools, with the exception of Look, end with a question mark to signify that they ask the object something. Figure 12 shows how each class of objects responds to questions about other objects. The Look tool differs from all other tools in that it can not be used by itself. Looks reveals the name of each object given to it by another inspection tool. Without Look, all you wil' see is an unintelligible machine code answer.

Objects / Inspection Tools	Node	Link	Channel	Circuit
Nodes?	If alive, returns itself as an object.	Returns the nodes, if alive, at either link end.	Returns the nodes, if alive, at either channel end.	Returns all living nodes the circuit passes through.
Links?	Returns all living links owned by a node.	If alive, returns itself as an object.	Returns the link, if alive, that carries the channel.	Returns all living links the circuit passes through.
Channels?	Returns all living channels owned by a node.	Returns all living channels owned by a link.	If alive, returns itself as an object.	Returns all living channels the circuit passes through.
Circuits?	Returns all active circuits that pass through a node.	Returns all living circuits owned by a link.	Returns the circuit, if allocated, or spare if not.	If alive, returns itself as an object.
Requests?	Returns the messages stored in REQUESTS.			
Replies?	Returns the messages stored in REPLIES			

Figure 12. Inspection Tools

Completion

To exit a PRP session first reset the display to text mode by typing

(NORMAL)

then to return to DOS enter

(EXIT) .

Source Code

Classes

Methods

Commands

Inspection Tools

Utilities

Graphics

System Files and Macros

Networks

```
;*****************************************************************************
;*            PROTOTYPE ROUTE PLANNER FOR ZENITH 248
;*
;*    DATE: 1 September 1988
;*  VERSION: 1.0
;*    TITLE: Class Declarations
;* FILENAME: class.s
;*
;* OPER SYS: DOS VERSION 3.2
;* LANGUAGE: PC Scheme (Ver 3.0 Student Edition) w/SCOOPS
;*   AUTHOR: Capt G.R. Gier
;*
;*  CONTENTS
;*      CLASSES: IDENTITY, NODE, LINK, CHANNEL, CIRCUIT
;*      METHODS: living, auto-connect-Circuit, one-more
;*     COMMANDS: none
;*        TOOLS: none
;*    UTILITIES: none
;*    SYS/MACRO: none
;*
;* CALLED BY: none
;*    CALLS: none
;*
;*   INPUTS: New-Circuit, Channel
;*  OUTPUTS: none
;*
;*   VARIABLES USED: Status, Circuit, Name
;* VARIABLES CHANGED: Channels, Status
;*      FILES READ: none
;*    FILES WRITTEN: none
;*
;*
;* FUNCTION:
;*
;* Nodes, links, channels, and circuits represent the four major classes of
;* objects in this environment.  An additional class called Identity is mixed
;* in to provide an initiable name-value upon creation and a status value
;* (dead or alive) that can change during program execution.  In the CNCE
;* world, a strict hierarchy prevents one object from talking to any other.
;* The relationship looks like this, with arrows representing communication
;* flow:  Node <---> Link <---> Channel <---> Circuit .

;*****************************************************************************


(define-class IDENTITY
        (instvars Name
                (Status 'Alive))
        (options (gettable-variables Name)
                (settable-variables Status)
                (inittable-variables Name)))
```

```
(define-method (IDENTITY living) ()                    ;Determines if the
        (if (eq? Status 'Alive) #T #F))          ;object will respond.

(compile-class IDENTITY)              ;Enters the class into the computer.

(define-class NODE
        (classvars (Population 0))
        (mixins IDENTITY)
        (instvars (Rectangle (make-window #F #T))
              (Serial-Number (set! Population (1 + Population)))
              Requests              ;A table of request messages.
              Replies               ;A table of reply messages.
              Links)            ;A list of links that node owns.
        (options (gettable-variables Requests Replies Links)
             (settable-variables Requests Replies Links Population)))

(compile-class NODE)

(define-class LINK
        (mixins IDENTITY)
        (instvars (Score-Board (make-window #F #F))
              Nodes             ;A list of nodes the link owns.
              Channels)         ;A list of channels the link owns.
        (options gettable-variables
             settable-variables
             (inittable-variables Nodes)))

(compile-class LINK)

(define-class CHANNEL
        (mixins IDENTITY)
        (instvars Link              ;The link that owns the channel.
              (Circuit (active 'Spare #F auto-connect-Circuit)))
                              ;The circuit allocated to the
                              ;channel, or 'Spare if free.
        (options gettable-variables
             settable-variables
             (inittable-variables Link)))

(define-method (CHANNEL auto-connect-Circuit) (New-Circuit)
         (if (eq? Circuit 'Spare)    .      ;Check if a spare channel.
            (cond ((eq? New-Circuit 'Spare) 'Spare)        ;No change.
                (else                    ;Allocate new circuit.
                  (send New-Circuit set-Channels (eval Name))
                  New-Circuit))
            (cond ((eq? New-Circuit 'Spare)
                (send Circuit set-Status 'Dead)     ;Remove channel.
                'Spare)                          ;Set to spare.
                (else
                  (send Circuit set-Status 'Dead)    ;Remove channel.
                  (send New-Circuit set-Channels (eval Name))
                  New-Circuit))))            ;Allocate new circuit.
```

```
(compile-class CHANNEL)

(define-class CIRCUIT
        (mixins IDENTITY)
        (instvars (Channels (active '() #F one-more))  ;A channel list.
              (Priority-Cost (vector 0 0 0 1)))   ;Priority vector.
        (options gettable-variables            ;form #( A B C Spare).
             settable-variables
             inittable-variables))

(define-method (CIRCUIT one-more) (Channel)
         (if (member Channel Channels) (delete! Channel Channels)
            (append Channels (list Channel))))

(compile-class CIRCUIT)

;*****************************************************************************
;*                                                                           *
;*                          END                                              *
;*                                                                           *
;*****************************************************************************
```

```
;*****************************************************************************
;*            PROTOTYPE ROUTE PLANNER FOR ZENITH 248
;*
;*    DATE: 2 September 1988
;*  VERSION: 1.0
;*    TITLE: Procedural Methods for Relaying Messages in Nodes
;* FILENAME: relay.s
;*
;* OPER SYS: DOS VERSION 3.2
;* LANGUAGE: PC Scheme (Ver 3.0 Student Edition) w/SCOOPS
;*   AUTHOR: Capt G.R. Gier
;*
;*  CONTENTS
;*      CLASSES: Node
;*      METHODS: relay-it
;*     COMMANDS: none
;*        TOOLS: none
;*    UTILITIES: none
;*    SYS/MACRO: none
;*
;* CALLED BY: relay
;*    CALLS: draw, exclude, living, look, relay, sort-links,
;*           table-lookup, tack-on, v>
;*
;*   INPUTS: Message
;*  OUTPUTS: Message
;*
;*   VARIABLES USED: Requests
;* VARIABLES CHANGED: Requests
;*      FILES READ: none
;*    FILES WRITTEN: none
;*
;*
;* FUNCTION:
;*
;* The node version of Relay-It acts as a main program in that it determines
;* which subordinate modules to call. Relay-It first checks for termination
;* of a request. If the node matches the message destination, it checks its
;* Replies table for previous messages. If a duplicate exists, Relay-it
;* compares the cost of the stored message with that of the new. If the new
;* route costs less, it is incorporated into the Replies table during lookup,
;* otherwise it is discarded. Whether a first time message or a cheaper,
;* duplicat message, Relay-It beeps signifying success without halting
;* subsequent message passing. Just like the popping of popcorn, you check
;* the results (using pr) when the noise stops. Furthermore, if you wait too
;* long the product burns as subsequent request messages flush the Replies
;* table.
;* --continued--
;*****************************************************************************
(define Light-Magenta 61)

(define Magenta 5)
```

```
(define-method (NODE relay-it) (Message)
 (draw Light-Magenta)
 (if (living)
   (let* ((Circuit-ID (list-ref Message 0))
        (Mode (list-ref Message 1))
        (Destination (eq? (list-ref Message 2) Name))
        (Priority (list-ref Message 3))
        (Cost (list-ref Message 4))
        (Path (list-ref Message 5))
        (Origin (eq? Name (car (last-pair Path))))
        (Beep (integer-char 7)))              ;Audio alert message.
     (cond ((eq? Mode 'request)               ;A Request message.
         (let* ((Dup-Request (table-lookup 'requests Message))
            ;Table-Lookup returns duplicate message or '().
            (Previous-Cost (list-ref Dup-Request 4))
                 ;Cost of the stored message.
            (Lower-Cost (v> Previous-Cost Cost))
                 ;Returns true or false.
            (From-Link (cadr Path))
                 ;The request came via this link.
            (Ordered-Links
             (exclude (eval From-Link) (sort-links Priority))))
                 ;An ordered list of which link to talk to.
          (cond (Destination ;Terminate message if the destination.
               (let* ((Dup-Reply;Returns stored message or '().
                    (table-lookup 'replies Message))
                 (Cost-Paths (car (last-pair Dup-Reply)))
                      ;((Cost-1 Path-1)...(Cost-N Path-N)).
                 (Previous-Cost (caar (last-pair Cost-Paths)))
                      ;Cost-N.
                 (Expensive-Route (v> Cost Previous-Cost)))

                      ;True or false.
               (writeln Name ": I am the message destination.")
               (if Dup-Reply
                 (if Expensive-Route
                   (begin
                     (writeln Name ": Too expensive--" Message)
                       ;Message terminates.
                     (draw Magenta))
                   (begin
                     (append! Cost-Paths
                       ;Add an alternative route to the others.
                          (list (list Cost Path)))
                     (writeln Name ": Include an alternate
route--"
                          Message)
                     (draw Magenta)
                     (display Beep))) ;Signal success.
                 (begin (writeln Name ": New route--" Message)
                      ;First time message.
```

```
                    (draw Magenta)
                    (display Beep))))) ;Signal success.
;(*****************************************************************************
;* If not the destination, Relay-It checks if it has previously passed the
;* same message. If a lower cost duplicate resides in the Requests
;* table, the message expires. If the new message is cheaper, Relay-It
;* replaces the old with the new and then passes the new message on to
;* all living links except the one that delivered the message. Before the
;* link hand-off however, Relay-It tacks on the link's name to the path list.
;* If Relay-It can't find a duplicate request message (first time message),
;* it annotates Requests via Table-Lookup, modifies the path and relays the
;* new message to the appropriate links.
;*
;* --continued--
;*****************************************************************************)
            (Dup-Request
                ;Transient message, check for previous passage.
             (writeln Name ": I already have this message.")
             (if Lower-Cost
                ;If cheaper, replace old message and re-transmit.
               (begin
                (writeln Name ": Its cheaper so keep it.")
                (set-car! (member Dup-Request Requests) Message)
                                ;Swap old for new.
                (writeln Name ": Update my friends "
                      (look Ordered-Links))
                (draw Magenta)
                (for-each (lambda (Link)     ;Update links.
                        (Relay Link (tack-on Link Message)))
                       Ordered-Links))
               (begin
                (draw Magenta)
                (writeln Name ": Message too expensive to
pass."))))
            (else ;First time transitory message.
             (writeln Name ": I have a new message.")
             (writeln Name ": Tell my friends " (look
Ordered-Links))
             (draw Magenta)
             (for-each (lambda (Link)        ;Tell links.
                    (Relay Link (tack-on Link Message)))
                   Ordered-Links)))))
;(*****************************************************************************
;* If Relay-It receives a Reply message, the node routes the message to the
;* next link in the Path for channel allocation, that is unless Relay-It
;* corresponds with the message origin, in which case the method terminates.
;* Note that the back link is to the right of the Relay-It node, which is
;* one link closer to the origin.
;*
;* Path: (Destination Dest-Side-Link Relay-It-Node Origin-Side-Link Origin)
;*
;*****************************************************************************)
```

```
        ((eq? Mode 'reply)
         (let ((Back-Link (cadr (member Name Path))))
          (cond ((not Origin)
                    ;Terminate if the origin, else pass message.
              (writeln Name ": Pass reply message to "
                    (look Back-Link))
              (draw Magenta)
              (relay (eval Back-Link) Message))
             (else
               (draw Magenta)
               (writeln "Attention! " Circuit-ID " now exits via "
                    Path)))))

;(*****************************************************************************
;* If the Relay-It node receives a Reject message and it is also the message
;* destination, it sounds a warning beep, signaling the tech controller to
;* pick a new route using PR.  If not the destination, the node relays the
;* message to the destination side link.
;*****************************************************************************)

        ((eq? Mode 'reject)
         (let ((Previous-Link (cadr (member Name (reverse Path))))
             (Processed-Reply (table-lookup 'replies Message)))
                         ;Returns stored message or '().
          (cond ((not Destination)      ;Pass the message.
              (writeln Name ": Send reject message to "
                    (look Previous-Link))
              (draw Magenta)
              (relay (eval Previous-Link) Message))
             (else              ;Annotate bad route and warn.
               (let* ((Bad-Route-Index
                     (sub1 (list-ref Processed-Reply 1)))
                    (Cost-Paths
                     (reverse (car (last-pair Processed-Reply))))
                          ;((Cost-1 Path-1)...(Cost-N Path-N)).
                    (Bad-Route
                     (list-ref Cost-Paths Bad-Route-Index)))
                (set-car! (cdr Processed-Reply) '())
                (set-car! Bad-Route "---Bad--- ")
                (draw Magenta)
                (do ((n 0 (1+ n))) ;Triple beep warning message.
                   (( n 3) (writeln Name ": Pick a new route for "
                          Circuit-ID))
                    (display Beep))))))))))
;*****************************************************************************
;*                                                                           *
;*                          END                                              *
;*                                                                           *
;*****************************************************************************
```

```
;*****************************************************************************
;*            PROTOTYPE ROUTE PLANNER FOR ZENITH 248
;*
;*    DATE: 17 August 1988
;*  VERSION: 1.0
;*   TITLE: Procedural Methods for Relaying Messages in Links
;* FILENAME: relay.s
;*
;* OPER SYS: DOS VERSION 3.2
;* LANGUAGE: PC Scheme (Ver 3.0 Student Edition) w/SCOOPS
;*  AUTHOR: Capt G.R. Gier
;*
;*  CONTENTS
;*      CLASSES: Link
;*      METHODS: relay-it
;*     COMMANDS: none
;*        TOOLS: none
;*    UTILITIES: none
;*    SYS/MACRO: none
;*
;* CALLED BY: relay
;*    CALLS: coord, cost?, draw, living, look, relay, select-channel
;*           table-lookup, tack-on, v>, v+
;*
;*   INPUTS: Message
;*  OUTPUTS: Message
;*
;*   VARIABLES USED: Nodes, Channels
;* VARIABLES CHANGED: none
;*      FILES READ: none
;*    FILES WRITTEN: none
;*
;*
;* FUNCTION:
;*
;* The Link version of Relay-It passes messages, examines capacity, and
;* allocates resources. When Relay-It picks up a Receive message and its
;* cost justifies preempting a channel, then Relay-It adds that cost
;* to the message, updates the path, and hands the information over to the
;* opposite end node. Note, no allocation takes place in this mode.
;*
;* --continued--
;*****************************************************************************)

(define-method (LINK relay-it) (Message)
 (if (living)
    (let* ((Circuit-ID (list-ref Message 0))
           (Mode (list-ref Message 1))
           (Dest (list-ref Message 2))
           (Priority (list-ref Message 3))
           (Old-Cost (list-ref Message 4))
           (Path (list-ref Message 5))
```

```
      (Origin-Side-Node (eval (cadr (member Name Path))))
      (Dest-Side-Node (if (eq? Origin-Side-Node (car Nodes))
                  (cadr Nodes)
                  (car Nodes)))
      (Chosen-Channel (car (sort-channels)))
      (Preempt-Cost (cost? Chosen-Channel))
      (Orig-Coord (send Origin-Side-Node coord))
      (Dest-Coord (send Dest-Side-Node coord)))
    (draw 'Light-Green Orig-Coord Dest-Coord '())
    (cond ((eq? Mode 'request)
        (if (v> Priority Preempt-Cost) ;Check for a usable channel.
          (let* ((New-Cost (v+ (list Old-Cost Preempt-Cost)))
                    ;Update the cost of using the channel.
              (New-Message (append (list Circuit-ID)
                    ;Reconstruct the message.
                        (list Mode)
                        (list Dest)
                        (list Priority)
                        (list New-Cost)
                        (list Path))))
           (writeln Name ": I have an allocatable channel--"
                (look Chosen-Channel))
           (draw 'Green Orig-Coord Dest-Coord (cost? (eval Name)))
           (relay Dest-Side-Node ;Send updated message.
                (tack-on Dest-Side-Node New-Message)))
          (begin
           (draw 'Green Orig-Coord Dest-Coord (cost? (eval Name)))
           (writeln Name ": I dont have a channel."))))


;*****************************************************************************
;* Allocation takes place in the Reply mode, providing Chosen-Channel picks
;* a preemptable channel. If not, Relay-It changes the message mode to
;* Reject and reverses path direction as it hands back the message to the
;* issuing node. Relay-It must make a second channel check since other
;* concurrent messages may steal the resource between the initial request
;* message and subsequent allocation by the reply message. Successful
;* allocation means the channel picks up a new circuit (and kills the old
;* circuit if necessary). Likewise, the circuit gathers a new channel into
;* its list of channels.
;*****************************************************************************

      ((eq? Mode 'reply)
       (let ((Distress-Message '(,Circuit-ID
                    Reject
                    ,@(list-tail Message 2))))
        (if (v> Priority Preempt-Cost)
                    ;Can't find a channel for some reason.
          (begin (writeln Name ": Allocate " (look Chosen-Channel)
                  " to " Circuit-ID)
              (send Chosen-Channel set-Circuit (eval Circuit-ID))
```

```
                ;Give channel circuit.
            (draw 'Red
                Dest-Coord Orig-Coord (cost? (eval Name)))
            (relay Origin-Side-Node Message))
                ;A-OK, pass message.
        (begin (writeln Name ": I lost my chosen channel.  Help!")
            (draw 'Green
                Orig-Coord Dest-Coord (cost? (eval Name)))
            (relay Dest-Side-Node Distress-Message)))))
                ;Don't allocate and pass back distress.

;*****************************************************************************
;
;* When Relay-It receives a Reject message, it de-allocates its channel
;* from the circuit and passes the information back towards the destination.
;*****************************************************************************
;

    ((eq? Mode 'reject)
     (let ((Undo-Channel (select-channel Circuit-ID Channels)))
      (writeln Name ": Free " (look Undo-Channel) ".")
      (send Undo-Channel set-Circuit 'Spare)     ;Free the channel.
      (draw 'Green Orig-Coord Dest-Coord (cost? (eval Name)))
      (relay Dest-Side-Node Message)))))))      ;Pass the message.

(define (relay Object Message)        ;Generic function for calling Relay-It.
 (send Object relay-it Message))      ;Works for NODE and LINK versions.

;*****************************************************************************
;
;*                                    *
;
;*                END                    *
;*                                    *
;
;*****************************************************************************
;
```

```
;*****************************************************************************
;*            PROTOTYPE ROUTE PLANNER FOR ZENITH 248
;*
;*    DATE: 17 August 1988
;*  VERSION: 1.0
;*   TITLE: Request-Table and Reply-Table lookup of transiting messages.
;* FILENAME: table_lu.s
;*
;* OPER SYS: DOS VERSION 3.2
;* LANGUAGE: PC Scheme (Ver 3.0 Student Edition) w/SCOOPS
;*   AUTHOR: Capt G.R. Gier
;*
;*  CONTENTS
;*      CLASSES: NODE
;*      METHODS: table-lookup
;*     COMMANDS: none
;*        TOOLS: none
;*    UTILITIES: none
;*    SYS/MACRO: none
;*
;* CALLED BY: relay-it
;*    CALLS: undup
;*
;*   INPUTS: Mode, Message
;*  OUTPUTS: stored Message from Request or Replies, or #F
;*
;*   VARIABLES USED: Requests, Replies
;* VARIABLES CHANGED: Requests, Replies
;*      FILES READ: none
;*    FILES WRITTEN: none
;*
;*
;* FUNCTION:
;*
;* This method services a node's Requests and Replies tables when called on
;* by the method Relay-It. Table-Lookup checks to see if the message that
;* was passed to it already exists in the appropriate table as determined by
;* the value of Mode. 1) If the table has no values (first message),
;* Table-Lookup enters the message as a list element. 2) If the message
;* does not match any existing messages, it is appended to the end of the
;* current list. If the new list exceeds five messages (arbitrary length
;* choice), Table-Lookup trims the oldest message effectively creating a
;* FIFO stack. In both the empty and no-match conditions, Table-Lookup
;* returns #F as its value. 3) If Table-Lookup finds a message,
;* it returns that message. Notice Table-Lookup modifies the
;* structure of messages in the Replies table. This modification permits
;* later inclusion of alternative path routes by Circuit-ID.
;*
;*****************************************************************************

(define-method (NODE table-lookup) (Mode Message)
 (let ((Table (eval Mode))          ;'Requests or 'Replies.
```

```
    (Circuit-ID (list-ref Message 0))
    (Selection '())              ;Place holder for later route choice.
    (Dest (list-ref Message 2))
    (Priority (list-ref Message 3))              ;Priority vector.
    (Unique (list-tail Message 4)))              ;(Cost Path).
  (cond ((eq? Table #!UNASSIGNED)                ;Empty table.
      (if (eq? Mode 'Requests)
         (set! Requests (list Message))          ;Unmodified message.
         (set! Replies (list (list Circuit-Id    ;Modified message.
                             Selection
                             Dest
                             Priority
                             (list Unique)))))
      #F)                                        ;Return false.

      ;Check each element for a Circuit-ID match and return if found.
      ;Undup weeds out '()s and Car removes extra parens.
      ((car (undup (mapcar (lambda (X) (member Circuit-ID X)) Table))))

      ((eq? Mode 'Requests)
       (append! Requests (list Message))         ;Permanently add message.
       (if (> (length Table) 5)                  ;Trim if too long.
          (set! Requests (list-tail Table 1)))
       #F)                                       ;Return false.

      ((eq? Mode 'Replies)
       (append! Replies (list (list Circuit-ID   ;Permanently add
                        Selection                ;modified message.
                        Dest
                        Priority
                        (list Unique))))
       (if (> (length Table) 5)                  ;Trim if too long.
          (set! Replies (list-tail Table 1)))
       #F))))                                    ;Return false.

;*****************************************************************************
;*                                                                           *
;*                          END                                              *
;*                                                                           *
;*****************************************************************************
```

```
;*****************************************************************************
;*            PROTOTYPE ROUTE PLANNER FOR ZENITH 248
;*
;*    DATE: 24 August 1988
;*  VERSION: 1.0
;*    TITLE: Link and channel sort routines
;* FILENAME: sort.s
;*
;* OPER SYS: DOS VERSION 3.2
;* LANGUAGE: PC Scheme (Ver 3.0 Student Edition) w/SCOOPS
;*   AUTHOR: Capt G.R. Gier
;*
;*  CONTENTS
;*      CLASSES: LINK, NODE
;*      METHODS: sort-channels, sort-links
;*     COMMANDS: none
;*        TOOLS: none
;*    UTILITIES: examine-link, useable
;*    SYS/MACRO: none
;*
;* CALLED BY: relay-it
;*    CALLS: cost?, examine-link, living, preempt+, useable, v>
;*
;*   INPUTS: Priority
;*  OUTPUTS: A channel list; A link list
;*
;*   VARIABLES USED: Channels, Links
;* VARIABLES CHANGED: none
;*      FILES READ: none
;*    FILES WRITTEN: none
;*
;*
;* FUNCTION:
;*
;* This module contains methods for the Node and Link class of objects.
;* Although the code in each is nearly identical, the results of calling
;* either method are different enough to warrant unique names for the
;* sort function. Sort-Links does what its name implies, it orders the
;* links that a node owns so that the first link in the returned list is the
;* link with the greatest amount of usable resources. The remaining links
;* have decreasingly fewer resources. If sort-link is passed a dead link,
;* or one that can't honor a channel request, it excludes it from the list.
;* This sort process, or heuristic, says the link with greatest reserve
;* is the one with the best chance of successfully completing a circuit.
;*
;* Sort-Channels also does what its name says, it rank-orders channels so
;* that spare channels head the list, followed by C-priority, B-priority,
;* A-priority, and lastly dead channels. Sort-Channels embodies a second
;* heuristic which says, allocate spare resources first, and then preempt
;* low priority channels before high priority channels.
;*
;*****************************************************************************
```

```
(define-method (LINK sort-channels) ()
        (if (living)
          (let ((Pairs (mapcar (lambda (Channel) ;Pair a channel
                               (list        ;to its cost.
                                (preempt+ (cost? Channel))
                                Channel))
                               Channels)))
            (mapcar cadr (sort! Pairs v>)))))   ;Sort by cost and
                                                ;return channel list.

(define-method (NODE sort-links) (Priority)  ;Returns an ordered list of links
        (if (living)              ;acceptable for message priority.
          (let ((Pairs (examine-link Priority Links)))
            (mapcar cadr (sort! Pairs v>)))))   ;Sort by capacity.

(define (examine-link Priority Links)       ;Returns a list of pairs.
 (if (null? Links) nil                  ;(capacity-vector link)
   (let* ((Preempt-Cost (preempt+ (cost? (car Links))))
          (Valid (useable Priority Preempt-Cost)))
     (if Valid
       (append '((,Valid ,(car Links)))
               (examine-link Priority (cdr Links)))
       (examine-link Priority (cdr Links))))))

(define (useable Priority Preempt-Cost)     ;Determines how many channels
 (do ((Result (vector 0 0 0 0))             ;can satisfy a given priority
      (Done #F)                             ;and returns that number.
      (Index 1 (1+ Index)))
     ((> Index 3) (if Done Result))
     (let ((Supply (vector-ref Preempt-Cost Index))
           (Demand (vector-ref Priority (sub1 Index))))
       (if Done (vector-set! Result Index Supply)
           (if (positive? Demand)
               (if (>= Supply Demand)
                   (begin
                    (vector-set! Result Index Supply)
                    (set! Done #T))))))))

;*****************************************************************************
;*                                                                           *
;*                              END                                          *
;*                                                                           *
;*****************************************************************************
```

```
;*****************************************************************************
;*            PROTOTYPE ROUTE PLANNER FOR ZENITH 248
;*
;*    DATE: 23 August 1988
;*  VERSION: 1.0
;*   TITLE: Make-Circuit command
;* FILENAME: mc.s
;*
;* OPER SYS: DOS VERSION 3.2
;* LANGUAGE: PC Scheme (Ver 3.0 Student Edition) w/SCOOPS
;*  AUTHOR: Capt G.R. Gier
;*
;*  CONTENTS
;*      CLASSES: none
;*      METHODS: none
;*     COMMANDS: mc
;*        TOOLS: none
;*    UTILITIES: none
;*    SYS/MACRO: none
;*
;* CALLED BY: User
;*    CALLS: relay
;*
;*   INPUTS: Keyboard entry
;*  OUTPUTS: Message to origin node
;*
;*   VARIABLES USED: *Circuit-ID*
;* VARIABLES CHANGED: *Circuit-ID*
;*      FILES READ: none
;*    FILES WRITTEN: none
;*
;*
;* FUNCTION:
;*
;* The Make Circuit command takes the form: (mc Origin Destination Priority)
;* where Origin and Destination are numbers that identify a CNCE by its
;* suffix (the "1" in CNCE-1), and Priority is either 'a, 'b, or 'c. This
;* command creates a message that has a unique Circuit ID as its head. The
;* message is a list of symbols and vectors with the following format:
;*
;*       (Circuit-ID 'Reply Destination Prior-Vec Cost-Vec (Origin))
;*
;* Prior-Vec is the vector form of Priority. Priority 'a translates to
;* #(1 0 0 0); 'b, #(0 1 0 0); and 'c, #(0 0 1 0). Cost-Vec is the nul
;* vector #(0 0 0 0) since the circuit has yet to accumulate expenses for
;* traveling links. MC passes the message to the origin node for subsequent
;* re-transmission.
;*
;*****************************************************************************

(define (mc 1st 2nd 3rd) ()
 (let ((Origin (string->symbol          ;"CNCE-" prefix for origin.
```

```
        (string-append "CNCE-"
                (number->string 1st '(int)))))
    (Circuit-ID (string->symbol
            (string-append
             "CIRCUIT-" (number->string
                    (setl *Circuit-ID* (1 + *Circuit-ID*))
                    '(int)))))
    (Mode 'request)          ;Determines how Relay treats message.
    (Dest (string->symbol
        (string-append "CNCE-" (number->string 2nd '(int)))))
    (Priority (case 3rd      ;Letter to vector.
            ((a) (vector 1 0 0 0))
            ((b) (vector 0 1 0 0))
            ((c) (vector 0 0 1 0))))
    (Cost (vector 0 0 0 0)))  ;Null vector.
  (relay (eval Origin)       ;Re-transmit message.
      (list Circuit-ID Mode Dest Priority Cost (list Origin))))
 "Message request sent.")

;*****************************************************************************
;*                                                                           *
;*                              END                                          *
;*                                                                           *
;*****************************************************************************
```

```
;*****************************************************************************
;*            PROTOTYPE ROUTE PLANNER FOR ZENITH 248
;*
;*    DATE: 21 August 1988
;*  VERSION: 1.0
;*   TITLE: Pick-Route command
;* FILENAME: pr.s
;*
;* OPER SYS: DOS VERSION 3.2
;* LANGUAGE: PC Scheme (Ver 3.0 Student Edition) w/SCOOPS
;*  AUTHOR: Capt G.R. Gier
;*
;*  CONTENTS
;*      CLASSES: none
;*      METHODS: none
;*     COMMANDS: pr
;*        TOOLS: none
;*    UTILITIES: pick-route, present
;*    SYS/MACRO: none
;*
;* CALLED BY: User
;*    CALLS: pick-route, present, relay
;*
;*   INPUTS: Keyboard
;*  OUTPUTS: Message
;*
;*   VARIABLES USED: Replies
;* VARIABLES CHANGED: Replies, Name, Priority-Cost
;*      FILES READ: none
;*    FILES WRITTEN: none
;*
;*
;* FUNCTION:
;*
;* The command Pick Route permits the technical controller at the destination
;* node to choose the appropriate route for implementation.  Pick Route
;* displays all unresolved requests one at a time, prompting for the route
;* choice.  Upon selection, PR generates a Reply message for execution
;* by the destination node and displays the next unresolved request if there
;* is one.  This command can also be used to select an alternative route if
;* the first choice is rejected.  The controller has a
;* choice of the lowest cost route, its equivalent cost alternates, and
;* high cost routes that are the first to reach the destination.  In other
;* words, the quantity of routes will vary, but the cheapest will always
;* fill the number one position.
;*
;*****************************************************************************


(define menu (make-window "Pick Route" #T))

(window-set-position! menu 4 15)
```

```
(window-set-size! menu 17 50)

(define (pr Node-Number)    ;Format: (pr 1).
 (let* ((Node (eval
            (string->symbol
             (string-append
              "CNCE-" (number->string Node-Number '(int))))))
       (Reply-Table (send Node get-Replies)))
  (for-each pick-route Reply-Table)) ;Sequentially display unresolved routes
 "No more requests.")

(define (pick-route Route-Choices)
 (let* ((Circuit-ID (car Route-Choices))
       (Picked (cadr Route-Choices))
       (Dest (caddr Route-Choices))
       (Priority (cadddr Route-Choices))
       (Pairs (reverse (list-ref Route-Choices 4)))
       (Max (length Pairs)))
  (cond ((not Picked)          ;Unresolved route.
        (window-popup menu)
        (display "This is " menu)
        (display Dest menu)
        (display " responding to new circuit requests." menu)
        (display #\newline menu)
        (display #\newline menu)
        (display "Possible routes for: " menu)
        (display Circuit-ID menu)
        (display #\newline menu)
        (display #\newline menu)
        (display " Number   Cost      Route" menu)  ;Header information.
        (set! Picked (present 1 Pairs))
                          ;List alternatives and return the choice.
        (cond ((eq? Picked 'a)
             (window-popup-delete menu))          ;Abort.
            ((> Picked Max) (pick-route Route-Choices)) ;Erroneous entry.
            (else
             (set-car! (member '() Route-Choices) Picked);Remember choice.
             (eval '(define ,Circuit-ID
                   (make-instance CIRCUIT
                            'Name ',Circuit-ID
                            'Priority-Cost ,Priority)))
                                   ;Create new circuit.
             (window-popup-delete menu)
             (relay (eval Dest)
                  (list Circuit-ID 'Reply Dest Priority
                                   ;Build reply message.
                       (car (list-ref Pairs (sub1 Picked)))
                         ;Subtract one since 0 represents the first.
                       (cadr (list-ref Pairs (sub1 Picked)))))))))))

(define (present Index Pairs) ;A utility to display cost-path pairs.
```

```
(cond ((null? Pairs)
     (display #\newline menu)
     (display #\newline menu)
     (display "Enter route number or A to abort." menu)
     (read))          ;Read route number choice.
    (else
     (display #\newline menu)
      (display "   " menu)
      (display Index menu)
      (display "." menu)
      (display #\tab menu)
      (display (caar Pairs) menu)
      (display "  " menu)
      (display (cadar Pairs) menu)
      (present (1 + Index) (cdr Pairs)))))
```

```
;*****************************************************************************
;*                                                                           *
;*                          END                                              *
;*                                                                           *
;*****************************************************************************
```

```
;*****************************************************************************
;*            PROTOTYPE ROUTE PLANNER FOR ZENITH 248
;*
;*    DATE: 17 August 1988
;*  VERSION: 1.0
;*   TITLE: The life and death of objects
;* FILENAME: life.s
;*
;* OPER SYS: DOS VERSION 3.2
;* LANGUAGE: PC Scheme (Ver 3.0 Student Edition) w/SCOOPS
;*  AUTHOR: Capt G.R. Gier
;*
;*  CONTENTS
;*     CLASSES: none
;*     METHODS: none
;*    COMMANDS: dead, alive
;*       TOOLS: living?
;*   UTILITIES: none
;*   SYS/MACRO: none
;*
;* CALLED BY: User
;*    CALLS: none
;*
;*  INPUTS: Object
;*  OUTPUTS: Status information
;*
;*  VARIABLES USED: Status
;* VARIABLES CHANGED: Status
;*     FILES READ: none
;*   FILES WRITTEN: none
;*
;*
;* FUNCTION:
;*
;* This module contains three commands used to check, and set the behavior
;* of nodes, links, channels, and circuits. If you ask an object if it is
;* living, it will reply with 'dead or 'alive. The second command, dead,
;* will "make dead" a single object or every object given to it in list
;* form. Similarly, the last command, alive, un-does the dead command.
;* Objects enter the environment alive, upon instantiation.
;*
;*****************************************************************************


(define (living? Object)
 (send Object living))

(define (dead Objects)
 (if (atom? Objects)
    (send Objects set-status 'dead)
    (mapcar (lambda (Object) (send Object set-status 'Dead)) Objects)))
```

```
(define (alive Objects)
 (if (atom? Objects)
    (send Objects set-status 'Alive)
    (mapcar (lambda (Object) (send Object set-status 'Alive)) Objects)))

;*****************************************************************************
;*                                                                           *
;*                          END                                              *
;*                                                                           *
;*****************************************************************************
```

```
;*****************************************************************************
;*            PROTOTYPE ROUTE PLANNER FOR ZENITH 248
;*
;*    DATE: 17 August
;*  VERSION: 1.0
;*   TITLE: Channel ownership inspection tool
;* FILENAME: channels.s
;*
;* OPER SYS: DOS VERSION 3.2
;* LANGUAGE: PC Scheme (Ver 3.0 Student Edition) w/SCOOPS
;*   AUTHOR: Capt G.R. Gier
;*
;*  CONTENTS
;*      CLASSES: none
;*      METHODS: owns-channels
;*     COMMANDS: none
;*        TOOLS: channels?
;*    UTILITIES: none
;*    SYS/MACRO: none
;*
;* CALLED BY: User
;*    CALLS: flatten, living, undup,
;*
;*   INPUTS: Object
;*  OUTPUTS: Machine representation of obects or nil.
;*
;*   VARIABLES USED: Links, Channels
;* VARIABLES CHANGED: none
;*      FILES READ: none
;*    FILES WRITTEN: none
;*
;*
;*  FUNCTION:
;*
;* Nodes, links, circuits, and even channels can reply with a list of
;* channels that they own. The method call in each case is owns-channels.
;* If you ask a channel if it owns itself, it will reply with an environment
;* representation if the mix-in instance variable Status is set to 'Alive.
;* If Status is 'Dead, the response is '(). The function "channels?" applies
;* to all four classes of objects and simplifies the query process. For
;* example "(channels? CNCE-1)" returns a list of living channels.
;*
;*
;*****************************************************************************

(define-method (NODE owns-channels) ()
        (if (living)
           (undup
             (flatten (mapcar channels? Links)))))

(define-method (LINK owns-channels) ()
        (if (living)
```

```
            (undup
             (mapcar channels? Channels))))

(define-method (CHANNEL owns-channels) ()
        (if (living) (eval Name)))

(define-method (CIRCUIT owns-channels) ()
        (if (living)
           (undup
             (mapcar channels? Channels))))

(define-method (NODE owns-channels) ()
        (if (living)
           (undup
             (flatten (mapcar channels? Links)))))

(define (channels? Object)
 (send Object owns-channels))

;*****************************************************************************
;*                                                                           *
;*                          END                                              *
;*                                                                           *
;*****************************************************************************
```

```
;*****************************************************************************
;*            PROTOTYPE ROUTE PLANNER FOR ZENITH 248
;*
;*    DATE: 16 August 1988
;*  VERSION: 1.0
;*    TITLE: Circuit ownership inspection tool
;* FILENAME: circuits.s
;*
;* OPER SYS: DOS VERSION 3.2
;* LANGUAGE: PC Scheme (Ver 3.0 Student Edition) w/SCOOPS
;*   AUTHOR: Capt G.R. Gier
;*
;*  CONTENTS
;*      CLASSES: none
;*      METHODS: owns-circuits
;*     COMMANDS: none
;*        TOOLS: circuits?
;*    UTILITIES: none
;*    SYS/MACRO: none
;*
;* CALLED BY: User
;*    CALLS: flatten, living, undup
;*
;*   INPUTS: Object
;*  OUTPUTS: Machine representation of obects or nil.
;*
;*   VARIABLES USED: Links, Channels, Circuit
;* VARIABLES CHANGED: none
;*      FILES READ: none
;*    FILES WRITTEN: none
;*
;*
;* FUNCTION:
;*
;* Nodes, links, channels, and even circuits can reply with a list of
;* circuits that they own. The method call in each case is owns-circuits.
;* If you ask a circuit if it owns itself, it will reply with an environment
;* representation if the mix-in instance variable Status is set to 'Alive.
;* If Status is 'Dead, the response is '(). The function "circuits?" applies
;* to all four classes of objects and simplifies the query process. For
;* example "(circuits? CNCE-1)" returns a list of living circuits that
;* transit CNCE-1.
;*
;*****************************************************************************

(define-method (NODE owns-circuits) ()
        (if (living)
           (undup
            (flatten
              (mapcar circuits? Links)))))

(define-method (LINK owns-circuits) ()
```

```
        (if (living)
          (undup
           (mapcar circuits? Channels))))

(define-method (CHANNEL owns-circuits) ()
        (if (living)
          (if (symbol? Circuit)
            Circuit
            (circuits? Circuit))))

(define-method (CIRCUIT owns-circuits) ()
        (if (living) (eval Name)))

(define (circuits? Object)
  (if (symbol? Object) Object
    (send Object owns-circuits)))

;*****************************************************************************
;*                                                                           *
;*                              END                                          *
;*                                                                           *
;*****************************************************************************
```

```
;*****************************************************************************
;*            PROTOTYPE ROUTE PLANNER FOR ZENITH 248
;*
;*    DATE: 16 August 1988
;*  VERSION: 1.0
;*    TITLE: How to find the cost of links and channels
;* FILENAME: cost.s
;*
;* OPER SYS: DOS VERSION 3.2
;* LANGUAGE: PC Scheme (Ver 3.0 Student Edition) w/SCOOPS
;*   AUTHOR: Capt G.R. Gier
;*
;*  CONTENTS
;*      CLASSES: none
;*      METHODS: cost-vector
;*     COMMANDS: none
;*        TOOLS: cost?
;*    UTILITIES: none
;*    SYS/MACRO: none
;*
;* CALLED BY: Keyboard, relay, sort
;*    CALLS: living, v+
;*
;*   INPUTS: Object
;*  OUTPUTS: Four element vector of numbers representing cost by priority
;*
;*   VARIABLES USED: Circuit, Channels
;* VARIABLES CHANGED: none
;*      FILES READ: none
;*    FILES WRITTEN: none
;*
;*
;* FUNCTION:
;*
;* This module contains the code that determines vector cost of a specific
;* channel or link. If a channel is free, the 'Spare value will cause a
;* value of #(0 0 0 1) to be returned depicting one spare in the format
;* #(A B C Spare). If a channel owns a circuit, the cost of preemptively
;* using the channel is the priority of the currently assigned circuit. If
;* the channel is dead it returns a vector nil, #(0 0 0 0). The cost of
;* using a link is the vector sum of the costs of channels it owns. A dead
;* link returns a vector nil, meaning it can not carry circuits.
;*
;*****************************************************************************


(define-method (CHANNEL cost-vector) ()
        (if (living)
           (if (symbol? Circuit)
              (vector 0 0 0 1)
              (if (circuits? Circuit)
                 (send Circuit get-Priority-Cost)
```

```
            (vector 0 0 0 1)))
        (vector 0 0 0 0)))

(define (cost? Object)
 (send Object cost-vector))

(define-method (LINK cost-vector) ()
      (if (living)
        (v+                          ;vector summation
         (mapcar cost? Channels))
        (vector 0 0 0 0)))

;*****************************************************************************
;*                                                                           *
;*                          END                                              *
;*                                                                           *
;*****************************************************************************
```

```
;*****************************************************************************
;*            PROTOTYPE ROUTE PLANNER FOR ZENITH 248
;*
;*    DATE: 16 August 1988
;*  VERSION: 1.0
;*   TITLE: Link ownership inspection tool
;* FILENAME: links.s
;*
;* OPER SYS: DOS VERSION 3.2
;* LANGUAGE: PC Scheme (Ver 3.0 Student Edition) w/SCOOPS
;*  AUTHOR: Capt G.R. Gier
;*
;*  CONTENTS
;*      CLASSES: none
;*      METHODS: owns-links
;*     COMMANDS: none
;*        TOOLS: links?
;*    UTILITIES: none
;*    SYS/MACRO: none
;*
;* CALLED BY: User
;*    CALLS: living, undup
;*
;*  INPUTS: Object
;*  OUTPUTS: Machine representation of objects or nil.
;*
;*  VARIABLES USED: Links, Channels
;* VARIABLES CHANGED: none
;*      FILES READ: none
;*    FILES WRITTEN: none
;*
;*
;* FUNCTION:
;*
;* Nodes, channels, circuits, and even links can reply with a list of
;* links that they own. The method call in each case is owns-links.
;* If you ask a link if it owns itself, it will reply with an environment
;* representation if the mix-in instance variable Status is set to 'Alive.
;* If Status is 'Dead, the response is '(). The function "links?" applies
;* to all four classes of objects and simplifies the query process. For
;* example "(links? CNCE-1)" returns a list of living links that terminate
;* at CNCE-1.
;*
;*****************************************************************************


(define-method (NODE owns-links) ()
        (if (living)
           (undup
            (mapcar links? Links))))

(define-method (LINK owns-links) ()
```

```
        (if (living) (eval Name)))

(define-method (CHANNEL owns-links) ()
        (if (living)
          (links? Link)))

(define-method (CIRCUIT owns-links) ()
        (if (living)
          (undup
            (mapcar links? Channels))))

(define (links? Object)
 (send Object owns-links))

;*****************************************************************************
;*                                                                           *
;*                              END                                          *
;*                                                                           *
;*****************************************************************************
```

```
;*****************************************************************************
;*            PROTOTYPE ROUTE PLANNER FOR ZENITH 248
;*
;*    DATE: 17 August 1988
;*  VERSION: 1.0
;*   TITLE: Machine code to proper name conversion tool
;* FILENAME: look.s
;*
;* OPER SYS: DOS VERSION 3.2
;* LANGUAGE: PC Scheme (Ver 3.0 Student Edition) w/SCOOPS
;*   AUTHOR: Capt G.R. Gier
;*
;*  CONTENTS
;*      CLASSES: none
;*      METHODS: none
;*     COMMANDS: none
;*        TOOLS: look
;*    UTILITIES: none
;*    SYS/MACRO: none
;*
;* CALLED BY: Keyboard, relay-it
;*    CALLS: none
;*
;*   INPUTS: Objects
;*  OUTPUTS: A single name or a list of object names
;*
;*   VARIABLES USED: Name
;* VARIABLES CHANGED: none
;*      FILES READ: none
;*    FILES WRITTEN: none
;*
;* FUNCTION:
;*
;* This tool permits the user to read the names of objects that are in
;* "environment" format.  Use look in conjunction with other tools.  For
;* example, if you want to know which links CNCE-1 owns, you would enter:
;*                    (look (links? CNCE-1))
;* The reply would appear something like:
;*                    (LINK-1 LINK-4 LINK-5)
;* Look filters out dead objects so the number of objects returned for
;* inspection, may not represent the number of objects in the original list.
;*****************************************************************************
(define (look Objects)
 (cond ((null? Objects) nil)              ;A dead object appears as '().
    ((symbol? Objects) Objects)           ;Permits 'Spare to be returned.
    ((atom? Objects) (send Objects get-name))   ;Retrieve object's name.
    (else (mapcar look Objects))))        ;Dead objects don't show in list.
;*****************************************************************************
;*                                                                           *
;*                          END                                              *
;*                                                                           *
;*****************************************************************************
```

```
;*****************************************************************************
;*            PROTOTYPE ROUTE PLANNER FOR ZENITH 248
;*
;*    DATE: 17 August 1988
;*  VERSION: 1.0
;*   TITLE: Node ownership inspection tool
;* FILENAME: node.s
;*
;* OPER SYS: DOS VERSION 3.2
;* LANGUAGE: PC Scheme (Ver 3.0 Student Edition) w/SCOOPS
;*  AUTHOR: Capt G.R. Gier
;*
;*  CONTENTS
;*      CLASSES: none
;*      METHODS: owns-nodes
;*     COMMANDS: none
;*        TOOLS: nodes?
;*    UTILITIES: none
;*    SYS/MACRO: none
;*
;* CALLED BY: User
;*    CALLS: flatten, living, undup
;*
;*  INPUTS: Object
;*  OUTPUTS: Machine representation of objects or nil.
;*
;*  VARIABLES USED: Channels, Link, Name, Nodes
;* VARIABLES CHANGED: none
;*      FILES READ: none
;*    FILES WRITTEN: none
;*
;*
;* FUNCTION:
;*
;* Channels, links, circuits, and even nodes can reply with a list of
;* nodes that they own. The method call in each case is owns-nodes.
;* If you ask a node if it owns itself, it will reply with an environment
;* representation if the mix-in instance variable Status is set to 'Alive.
;* If Status is 'Dead, the response is '(). The function "nodes?" applies
;* to all four classes of objects and simplifies the query process. For
;* example "(nodes? LINK-1)" returns a list of living nodes connected to
;* LINK-1.
;*
;*****************************************************************************

(define-method (NODE owns-nodes) ()
        (if (living) (eval Name)))

(define-method (LINK owns-nodes) ()
        (if (living)
           (undup
             (mapcar nodes? Nodes))))
```

```
(define-method (CHANNEL owns-nodes) ()
        (if (living)
           (undup
            (nodes? Link))))

(define-method (CIRCUIT owns-nodes) ()
        (if (living)
           (undup
            (flatten
             (mapcar nodes? Channels)))))

(define (nodes? Object)
 (send Object owns-nodes))

;*****************************************************************************
;*                                                                           *
;*                          END                                              *
;*                                                                           *
;*****************************************************************************
```

```
;*****************************************************************************
;*            PROTOTYPE ROUTE PLANNER FOR ZENITH 248
;*
;*    DATE: 17 August 1988
;*  VERSION: 1.0
;*    TITLE: Reply table inspection tool
;* FILENAME: replies.s
;*
;* OPER SYS: DOS VERSION 3.2
;* LANGUAGE: PC Scheme (Ver 3.0 Student Edition) w/SCOOPS
;*  AUTHOR: Capt G.R. Gier
;*
;*  CONTENTS
;*      CLASSES: none
;*      METHODS: none
;*     COMMANDS: none
;*        TOOLS: replies?
;*    UTILITIES: none
;*    SYS/MACRO: none
;*
;* CALLED BY: User
;*    CALLS: display-replies, display-pairs
;*
;*  INPUTS: Node
;*  OUTPUTS: Formatted contents of the Replies Table.
;*
;*  VARIABLES USED: Replies
;* VARIABLES CHANGED: none
;*      FILES READ: none
;*    FILES WRITTEN: none
;*
;* FUNCTION:
;*
;* Replies? is an inspection tool that displays the request messages owned
;* by a node in an easy to read format. The node reply table retains its
;* original internal form as Replies? is not destructive.
;*****************************************************************************
(define (replies? Node)
 (for-each display-replies (send Node get-Replies)))    ;Gets reply table.

(define (display-replies Message)                       ;Prepare banner
 (writeln)                                  ;and common info.
 (writeln (car Message) " " (cadr Message) " " (caddr Message) " "
      (cadddr Message))
 (for-each display-pairs (list-ref Message 4)))         ;Display cost & path.

(define (display-pairs Pair)
 (writeln #\tab (car Pair) " " (cadr Pair)))            ;Indent for all routes
;*****************************************************************************
;*                        END                          *
;*                                                     *
;*****************************************************************************
```

```
;*****************************************************************************
;*          PROTOTYPE ROUTE PLANNER FOR ZENITH 248
;*
;*    DATE: 17 August 1988
;*  VERSION: 1.0
;*    TITLE: Request table inspection tool
;* FILENAME: requests.s
;*
;* OPER SYS: DOS VERSION 3.2
;* LANGUAGE: PC Scheme (Ver 3.0 Student Edition) w/SCOOPS
;*   AUTHOR: Capt G.R. Gier
;*
;*  CONTENTS
;*      CLASSES: none
;*      METHODS: none
;*     COMMANDS: none
;*        TOOLS: requests?
;*    UTILITIES: dis-req
;*    SYS/MACRO: none
;*
;* CALLED BY: User
;*    CALLS: dis-req
;*
;*   INPUTS: Node
;*  OUTPUTS: Formatted contents of Requests Table
;*
;*   VARIABLES USED: Requests
;* VARIABLES CHANGED: none
;*      FILES READ: none
;*    FILES WRITTEN: none
;*
;*
;* FUNCTION:
;*
;* This inspection tool displays messages in a node's request table, in an
;* easy to read format.  As a message enters the request table, it goes to
;* the end of the list.
;*
;*****************************************************************************
(define (requests? Node)
  (for-each dis-req (send Node get-Requests)))

(define (dis-req Message)
  (writeln)
  (writeln (car Message) " " (cadr Message) " " (caddr Message) " "
        (cadddr Message) " " (list-ref Message 4))
  (writeln #\tab (list-ref Message 5)))                ;Indent the path.
;*****************************************************************************
;*                                                   *
;*                    END                              *
;*                                                   *
;*****************************************************************************
```

```
;*****************************************************************************
;*          PROTOTYPE ROUTE PLANNER FOR ZENITH 248
;*
;*    DATE: 17 August 1988
;*  VERSION: 1.0
;*   TITLE: Exclude one object from a list of objects
;* FILENAME: exclude.s
;*
;* OPER SYS: DOS VERSION 3.2
;* LANGUAGE: PC Scheme (Ver 3.0 Student Edition) w/SCOOPS
;*  AUTHOR: Capt G.R. Gier
;*
;*  CONTENTS
;*      CLASSES: none
;*      METHODS: none
;*     COMMANDS: none
;*        TOOLS: none
;*    UTILITIES: exclude.s
;*    SYS/MACRO: none
;*
;* CALLED BY: relay-it
;*    CALLS: none
;*
;*  INPUTS: Object-key, List of Objects
;*  OUTPUTS: List of Objects without object-key.
;*
;*  VARIABLES USED: none
;* VARIABLES CHANGED: none
;*      FILES READ: none
;*    FILES WRITTEN: none
;*
;*
;* FUNCTION:
;*
;* This module excludes an object from a list of objects, returning a list
;* of the remaining objects. This module is called when (relay Node ...)
;* wants to pass a message to every link except the link that brought the
;* message.
;*
;*****************************************************************************

(define (exclude Object Lst)
 (cond ((null? Lst) nil)
     ((eq? Object (car Lst)) (exclude Object (cdr Lst)))
     (else (cons (car Lst) (exclude Object (cdr Lst))))))

;*****************************************************************************
;*                                    *
;*                END                   *
;*                                    *
;*****************************************************************************
```

```
;*****************************************************************************
;*            PROTOTYPE ROUTE PLANNER FOR ZENITH 248
;*
;*    DATE: 17 August 1988
;*  VERSION: 1.0
;*   TITLE: Paren remover
;* FILENAME: flatten.s
;*
;* OPER SYS: DOS VERSION 3.2
;* LANGUAGE: PC Scheme (Ver 3.0 Student Edition) w/SCOOPS
;*  AUTHOR: Capt G.R. Gier
;*
;*  CONTENTS
;*      CLASSES: none
;*      METHODS: none
;*     COMMANDS: none
;*        TOOLS: none
;*    UTILITIES: flatten
;*    SYS/MACRO: none
;*
;* CALLED BY: Channels, Circuits, Nodes
;*    CALLS: none
;*
;*   INPUTS: List
;*  OUTPUTS: List without muliple wrappings of parentheses
;*
;*   VARIABLES USED: none
;* VARIABLES CHANGED: none
;*      FILES READ: none
;*    FILES WRITTEN: none
;*
;*
;* FUNCTION:
;*
;* This utility removes multiple levels of parentheses, returning a proper
;* list of the original objects.  It is used to standardize the input for
;* other modules.
;*
;*****************************************************************************

(define (flatten Lst)
 (if (atom? Lst)
    (list Lst)
    (apply append (mapcar flatten Lst))))

;*****************************************************************************
;*                                                                           *
;*                          END                                              *
;*                                                                           *
;*****************************************************************************
```

```
;*****************************************************************************
;*          PROTOTYPE ROUTE PLANNER FOR ZENITH 248
;*
;*    DATE: 18 August 1988
;*  VERSION: 1.0
;*   TITLE: Gateway out of Scheme to DOS
;* FILENAME: out.s
;*
;* OPER SYS: DOS VERSION 3.2
;* LANGUAGE: PC Scheme (Ver 3.0 Student Edition) w/SCOOPS
;*  AUTHOR: Capt G.R. Gier
;*
;*  CONTENTS
;*     CLASSES: none
;*     METHODS: none
;*    COMMANDS: none
;*       TOOLS: none
;*    UTILITIES: out
;*    SYS/MACRO: none
;*
;* CALLED BY: User
;*    CALLS: none
;*
;*  INPUTS: none
;*  OUTPUTS: none
;*
;*  VARIABLES USED: none
;* VARIABLES CHANGED: none
;*      FILES READ: none
;*    FILES WRITTEN: none
;*
;*
;* FUNCTION:
;*
;* The utility Out temporarily suspends PC Scheme and gives you the DOS
;* prompt.  Out handily facilitates creation of fast load files (*.fsl)
;* as the executable make_fsl program does not work within PC Scheme.
;*
;*****************************************************************************

(define (out) ()
 (dos-call "" "" 4095))

;*****************************************************************************
;*                                                                           *
;*                          END                                              *
;*                                                                           *
;*****************************************************************************
```

```
;*****************************************************************************
;*            PROTOTYPE ROUTE PLANNER FOR ZENITH 248
;*
;*    DATE: 17 August 1988
;*  VERSION: 1.0
;*   TITLE: Determining the number of channels that could be preempted
;* FILENAME: preempt.s
;*
;* OPER SYS: DOS VERSION 3.2
;* LANGUAGE: PC Scheme (Ver 3.0 Student Edition) w/SCOOPS
;*  AUTHOR: Capt G.R. Gier
;*
;*  CONTENTS
;*      CLASSES: none
;*      METHODS: none
;*     COMMANDS: none
;*        TOOLS: none
;*    UTILITIES: preempt+, accum+
;*    SYS/MACRO: none
;*
;* CALLED BY: sort
;*    CALLS: accum+
;*
;*  INPUTS: Vector
;*  OUTPUTS: A modified vector that shows the number of available channels
;*         for each priority.
;*
;*  VARIABLES USED: none
;* VARIABLES CHANGED: none
;*      FILES READ: none
;*    FILES WRITTEN: none
;*
;*
;* FUNCTION:
;*
;* This utility takes a vector and cascade-sums from the lowest priority
;* to the highest priority. For example, preempt+ would operate on a
;* link cost vector of #(1 2 0 3) and return as output #(6 5 3 3).
;* The lowest priority or spare value is 3. Preempt+ retains 3 in the far
;* right column and adds it to 0 in the 'c priority column yielding 3 also.
;* Next, 2 from the 'b priority column is added to the previous sum 3,
;* giving 5. Finally the single 'a priority brings the far left column
;* total to 6. The resultant vector signifies that an 'a priority circuit
;* could use five 'b 'c or spare resources but no 'a priority resources.
;*
;*****************************************************************************

(define (preempt+ Vector);Format: #(A-prior B-prior C-prior Spare).
 (list->vector                    ;Convert to a list for recursion.
  (accum+                         ;Call list version.
   (vector->list Vector))))          ;Convert back to a vector.
```

```
(define (accum + Lst)
 (if (null? Lst) nil;If zero or '(), nil.
   (cons (apply + Lst) (accum + (cdr Lst)))))        ;Add all to the right.

;*************************************************************************
;*                                                                       *
;*                          END                                          *
;*                                                                       *
;*************************************************************************
```

```
;*****************************************************************************
;*            PROTOTYPE ROUTE PLANNER FOR ZENITH 248
;*
;*    DATE: 21 August 1988
;*  VERSION: 1.0
;*    TITLE: Selecting a channel allocated to a circuit from a channel list
;* FILENAME: sel_chan.s
;*
;* OPER SYS: DOS VERSION 3.2
;* LANGUAGE: PC Scheme (Ver 3.0 Student Edition) w/SCOOPS
;*   AUTHOR: Capt G.R. Gier
;*
;*  CONTENTS
;*      CLASSES: none
;*      METHODS: none
;*     COMMANDS: none
;*        TOOLS: none
;*    UTILITIES: select-channel
;*    SYS/MACRO: none
;*
;* CALLED BY: relay-it
;*    CALLS: select-channel
;*
;*   INPUTS: Circuit-ID, Channel list
;*  OUTPUTS: Machine code representation of channel
;*
;*   VARIABLES USED: Circuit
;* VARIABLES CHANGED: none
;*      FILES READ: none
;*    FILES WRITTEN: none
;*
;*
;* FUNCTION:
;*
;* The utility Select-Channel finds the channel that owns Circuit-ID from a
;* list of channels and returns the object Channel if successful, or false
;* if not.
;*
;*****************************************************************************

(define (select-channel Circuit-ID Channels)
 (if (null? Channels)
    #F
    (if (eq? (eval Circuit-ID) (send (car Channels) get-Circuit))
       (car Channels)
       (select-channel Circuit-ID (cdr Channels)))))

;*****************************************************************************
;*                                                                           *
;*                          END                                              *
;*                                                                           *
;*****************************************************************************
```

```
;*****************************************************************************
;*            PROTOTYPE ROUTE PLANNER FOR ZENITH 248
;*
;*    DATE: 17 August 1988
;*  VERSION: 1.0
;*   TITLE: Tack the name of a node or link to the beginning of a path
;* FILENAME: tack_on.s
;*
;* OPER SYS: DOS VERSION 3.2
;* LANGUAGE: PC Scheme (Ver 3.0 Student Edition) w/SCOOPS
;*   AUTHOR: Capt G.R. Gier
;*
;*  CONTENTS
;*      CLASSES: none
;*      METHODS: none
;*     COMMANDS: none
;*        TOOLS: none
;*    UTILITIES: tack-on
;*    SYS/MACRO: none
;*
;* CALLED BY: relay-it
;*    CALLS: none
;*
;*   INPUTS: Node or Link, and Message
;*  OUTPUTS: Updated path
;*
;*   VARIABLES USED: Name
;* VARIABLES CHANGED: none
;*      FILES READ: none
;*    FILES WRITTEN: none
;*
;* FUNCTION:
;*
;* The Tack-On utility inserts the symbolic representation of a link or node
;* at he beginning of the existing Path list.
;*
;*****************************************************************************
(define (tack-on Object Message)
 (let* ((Path (list-ref Message 5))
      (Name (if (symbol? Object) Object (send Object get-Name))) ;Get name.
      (New-Path (append (list Name) Path)))
   (list (list-ref Message 0)
      (list-ref Message 1)
      (list-ref Message 2)
      (list-ref Message 3)
      (list-ref Message 4)
      New-Path)))            ;Give back message with modified path.
;*****************************************************************************
;*                                    *
;*                END                   *
;*                                    *
;*****************************************************************************
```

```
;*****************************************************************************
;*            PROTOTYPE ROUTE PLANNER FOR ZENITH 248
;*
;*    DATE: 18 August 1988
;*  VERSION: 1.0
;*    TITLE: Remove duplicate elements from a list
;* FILENAME: undup.s
;*
;* OPER SYS: DOS VERSION 3.2
;* LANGUAGE: PC Scheme (Ver 3.0 Student Edition) w/SCOOPS
;*   AUTHOR: Capt G.R. Gier
;*
;*  CONTENTS
;*      CLASSES: none
;*      METHODS: none
;*     COMMANDS: none
;*        TOOLS: none
;*    UTILITIES: undup
;*    SYS/MACRO: none
;*
;* CALLED BY: owns-channels, owns-circuits, owns-links, owns-nodes, table-lookup
;*    CALLS: undup
;*
;*   INPUTS: List
;*  OUTPUTS: List with no duplicate elements
;*
;*   VARIABLES USED: none
;* VARIABLES CHANGED: none
;*      FILES READ: none
;*    FILES WRITTEN: none
;*
;*
;* FUNCTION:
;*
;* The utility Undup removes duplicate elements from a list, and removes
;* the '() left by dead objects and the 'Spare returned by channels,
;* presenting a list of living objects only.
;*
;*****************************************************************************

(define (undup Lst)
 (cond ((null? Lst) nil)                    ;Empty list, stop.
     ((null? (car Lst)) (undup (cdr Lst)))        ;Skip over '().
     ((symbol? (car Lst)) (undup (cdr Lst)))      ;Filter out 'Spare.
     ((member (car Lst) (cdr Lst)) (undup (cdr Lst)));Skip duplicate.
     (else (cons (car Lst) (undup (cdr Lst))))))    ;Build undup list.

;*****************************************************************************
;*                                                                           *
;*                          END                                              *
;*                                                                           *
;*****************************************************************************
```

```
;*****************************************************************************
;*            PROTOTYPE ROUTE PLANNER FOR ZENITH 248
;*
;*    DATE: 18 August 1988
;*  VERSION: 1.0
;*    TITLE: Determine if one vector is greater than another
;* FILENAME: vec_gt.s
;*
;* OPER SYS: DOS VERSION 3.2
;* LANGUAGE: PC Scheme (Ver 3.0 Student Edition) w/SCOOPS
;*   AUTHOR: Capt G.R. Gier
;*
;*  CONTENTS
;*      CLASSES: none
;*      METHODS: none
;*     COMMANDS: none
;*        TOOLS: none
;*    UTILITIES: v>
;*    SYS/MACRO: none
;*
;* CALLED BY: relay-it, sort
;*    CALLS: none
;*
;*   INPUTS: nulls or vectors
;*  OUTPUTS: boolean
;*
;*   VARIABLES USED: none
;* VARIABLES CHANGED: none
;*      FILES READ: none
;*    FILES WRITTEN: none
;*
;*
;* FUNCTION:
;*
;* V> compares the first vector to the second and returns true if greater
;* than the second and false if less than or equal to the second. If
;* handed the cost of a dead channel, '(), V> converts the nil into the
;* null vector and proceeds with the comparison. If wrapped in a list,
;* V> strips the list. Examples:
;* (v> #(1 0 0 0) #(0 1 0 0)) -> true.
;* (v> #(5 5 5 5) #(5 5 5 4)) -> true.
;* (v> #(0 9 8 7) #(1 0 0 0)) -> false.
;* (v> #(0 0 0 0) #(0 0 0 0)) -> false.
;*
;*****************************************************************************

(define (v> Object-1 Object-2)
  (let ((V-1 (cond ((vector? Object-1) Object-1)
                   ((null? Object-1) (vector 0 0 0 0))
                   (else (car Object-1))))
        (V-2 (cond ((vector? Object-2) Object-2)
                   ((null? Object-2) (vector 0 0 0 0))
```

```
            (else (car Object-2)))))
   (do ((Index 0 (1 + Index))        ;First iteration has no meaning.
      (First)
      (Second)
      (GT #F (> First Second))        ;Set GT flag to false, wait for change.
      (LT #F (< First Second)))       ;Set LT flag to false, wait for change.
     ((or GT LT (> Index 3)) GT)      ;Check flags, don't exceed vector
length.
      (set! First (vector-ref V-1 Index))          ;Check from left to right.
      (set! Second (vector-ref V-2 Index)))))

;*****************************************************************************
;*                                                                           *
;*                          END                                              *
;*                                                                           *
;*****************************************************************************
```

```
;*****************************************************************************
;*            PROTOTYPE ROUTE PLANNER FOR ZENITH 248
;*
;*    DATE: 18 August 1988
;*  VERSION: 1.0
;*   TITLE: Sum a list of vectors by element
;* FILENAME: vec_plus.s
;*
;* OPER SYS: DOS VERSION 3.2
;* LANGUAGE: PC Scheme (Ver 3.0 Student Edition) w/SCOOPS
;*  AUTHOR: Capt G.R. Gier
;*
;*  CONTENTS
;*      CLASSES: none
;*      METHODS: none
;*     COMMANDS: none
;*        TOOLS: none
;*     UTILITIES: v+
;*     SYS/MACRO: none
;*
;* CALLED BY: cost-vector, relay-it
;*    CALLS: none
;*
;*  INPUTS: List of vectors
;*  OUTPUTS: Vector
;*
;*  VARIABLES USED: none
;* VARIABLES CHANGED: none
;*      FILES READ: none
;*    FILES WRITTEN: none
;*
;*
;* FUNCTION:
;*
;* The utility V+ sums a list of vectors from right to left.  Note that no
;* vector units carry in the process.
;*
;*****************************************************************************
(define (v+ V-List)
 (do ((N 3 (-1+ N))                    ;A four place vector.
    (V-Sum '()                         ;A list accumulator.
        (cons (apply + (mapcar
                    (lambda (Vector) (vector-ref Vector N))
                    V-List))
              V-Sum)))
   ((negative? N) (list->vector V-Sum)))) ;Convert from list to vector.

;*****************************************************************************
;*                                                *
;*                    END                           *
;*                                                *
;*****************************************************************************
```

```
;*****************************************************************************
;*            PROTOTYPE ROUTE PLANNER FOR ZENITH 248
;*
;*    DATE: 2 September 1988
;*  VERSION: 1.0
;*   TITLE: Graphics module for nodes and links
;* FILENAME: draw.s
;*
;* OPER SYS: DOS VERSION 3.2
;* LANGUAGE: PC Scheme (Ver 3.0 Student Edition) w/SCOOPS
;*  AUTHOR: Capt G.R. Gier
;*
;*  CONTENTS
;*     CLASSES: none
;*     METHODS: coord, draw
;*    COMMANDS: none
;*       TOOLS: none
;*   UTILITIES: clear, clipper, list+, normal, x-ref-text, x-ref-graph,
;*              y-ref-text, y-ref-graph
;*   SYS/MACRO: none
;*
;* CALLED BY: relay-it
;*    CALLS: clipper, list+, living, normal, x-ref-text, x-ref-graph,
;*           y-ref-text, y-ref-graph
;*
;*  INPUTS: Color, From-node, To-node, Link-Cost
;*  OUTPUTS: Graphics
;*
;*  VARIABLES USED: Name
;* VARIABLES CHANGED: none
;*     FILES READ: none
;*   FILES WRITTEN: none
;*
;*
;* FUNCTION:
;*
;* Draw contains routines that permit the user to visually observe the search
;* and allocation processes. The first node is always draw at the top of the
;* screen, or 12 o'clock position. Additional nodes are drawn clockwise,
;* angularly spaced equi-distant from each other. The nodes are drawn using
;* text coordinates, but the links are drawn using graphics coordinates.
;* Clipper prevents a line from drawing over a node already on the screen.
;* The Clear and Normal utilities help reset the screen for graphics and text
;* modes respectively.
;*****************************************************************************

(define-method (NODE coord) ()
        (let* ((Angle-in-Radians (- (/ pi 2)
                                    (/ (* 2 pi (sub1 Serial-Number))
                                       Population)))
               (X (round (* 280 (cos Angle-in-Radians))))
               (Y (round (* 160 (sin Angle-in-Radians)))))
```

```
        (cons X Y)))

(define-method (NODE draw) (Color)
 (let* ((Node-Coord (coord))
        (Line (y-ref-text (cdr Node-Coord)))
        (Col (- (x-ref-text (car Node-Coord)) 4)))     ;Left, to center text.
  (window-set-position! Rectangle Line Col)
  (window-set-size! Rectangle 1 8)
  (cond ((living)      ;White is 15, Lgt-M is 61, Magenta is 5, Gray is 56.
        (window-set-attribute! Rectangle 'Border-Attributes Color)
        (window-set-attribute! Rectangle 'Text-Attributes 15))
        (else
        (window-set-attribute! Rectangle 'Border-Attributes 56)
        (window-set-attribute! Rectangle 'Text-Attributes 56)))
  (window-clear Rectangle)
  (display Name Rectangle)))

(define (x-ref-text X)
 (truncate (/ (+ X 320) 8)))

(define (y-ref-text Y)
 (truncate (/ (- 174 Y) 14)))

(define-method (LINK draw) (Color From To Link-Cost)
 (let* ((Node1 (send (car Nodes) coord))   ;Stored representation
        (Node2 (send (cadr Nodes) coord))  ;prevents dual labels
        (Pretty-Cost (if Link-Cost (vector->list Link-Cost)))
        (Clip1 (clipper From To))
        (Clip2 (clipper To From))
        (Clippings (list+ (list Clip1 Clip2)))
        (One-Third-X (round (x-ref-text
                     (/ (+ (* (car Node1) 2) (car Node2)) 3))))
        (One-Third-Y (round (y-ref-text
                     (/ (+ (* (cdr Node1) 2) (cdr Node2)) 3)))))
  (set-clipping-rectangle! (car Clippings)
                   (cadr Clippings)
                   (caddr Clippings)
                   (cadddr Clippings))
  (position-pen (car From) (cdr From))
  (cond ((living) (set-pen-color! Color)) ;Light-Green or Green.
       (else (set-pen-color! 'Gray)))
  (draw-line-to (car To) (cdr To))
  (set-clipping-rectangle! -320 174 319 -175)
  (cond (Link-Cost
        (window-set-position!
         Score-Board (sub1 One-Third-Y) (- One-Third-X 5))
        (window-set-size! Score-Board 2 10)
        (window-clear Score-Board)
        (display " " Score-Board)
        (display Name Score-Board)
        (display #\newline Score-Board)
        (display Pretty-Cost Score-Board))
```

```
      (else nil))))

(define (x-ref-graph X)
 (- (* (truncate (/ (+ X 320) 8)) 8) 320))

(define (y-ref-graph Y)
 (- (* (truncate (/ (+ Y 175) 14)) 14) 175))

(define (clipper Coord1 Coord2)            ;    G     F     E
 (let* ((x-of car)                    ;   ZDDDDDDDDDDDDDDDD?
      (y-of cdr)                      ;  3          3
      (X (x-ref-graph (x-of Coord1)))      ;   H3    X,Y    D3
      (Y (y-ref-graph (y-of Coord1)))      ;   3          3
      (A (cons (- X 39) (- Y 9)))          ;   @DDDDDDDDDDDDDDDDDY
      (C (cons (+ X 38) (- Y 9)))          ;   A     B     C
      (E (cons (+ X 38) (+ Y 21)))
      (G (cons (- X 39) (+ Y 21)))
      (Angle-A (atan (- (y-of A) (y-of Coord1)) (- (x-of A) (x-of Coord1))))
      (Angle-B (/ pi -2))
      (Angle-C (atan (- (y-of C) (y-of Coord1)) (- (x-of C) (x-of Coord1))))
      (Angle-D 0)
      (Angle-E (atan (- (y-of E) (y-of Coord1)) (- (x-of E) (x-of Coord1))))
      (Angle-F (/ pi 2))
      (Angle-G (atan (- (y-of G) (y-of Coord1)) (- (x-of G) (x-of Coord1))))
      (Angle-H pi)
      (Theta (atan (- (y-of Coord2) (y-of Coord1))
             (- (x-of Coord2) (x-of Coord1)))))
  (cond ((< Theta Angle-A) '(0      ,(y-of G) ,(x-of G) 0    ))
      ((< Theta Angle-B) '(0      ,(y-of C) ,(x-of C) 0    ))
      ((< Theta Angle-C) '(,(x-of A) ,(y-of A) 0      0    ))
      ((< Theta Angle-D) '(,(x-of E) ,(y-of E) 0      0    ))
      ((< Theta Angle-E) '(,(x-of C) 0      0      ,(y-of C)))
      ((< Theta Angle-F) '(,(x-of G) 0      0      ,(y-of G)))
      ((< Theta Angle-G) '(0      0      ,(x-of E) ,(y-of E)))
      (else          '(0      0      ,(x-of A) ,(y-of A))))))

(define (list+ Clippings)
 (do ((N 3 (-1 + N));A four place list.
    (Sum '();A list accumulator.
       (cons (apply + (mapcar
                (lambda (Lst) (list-ref Lst N))
                Clippings))
          Sum)))
   ((negative? N) Sum)))            ;Return totalled list.

(define (clear)
 (set-video-mode! 16)
 (window-set-position! 'console 0 0)
 (window-set-size! 'console 25 80)
 (window-clear 'console)
 (window-set-position! 'console 23 0)
 (window-set-size! 'console 2 35)
```

```
 (window-set-position! pcs-status-window 24 50)
 (window-set-size! pcs-status-window 1 30))

(define (normal)
 (set-video-mode! 3)
 (window-set-position! 'console 0 0)
 (window-set-size! 'console 24 80)
 (window-clear 'console))

;*****************************************************************************
;*                                                                           *
;*                          END                                              *
;*                                                                           *
;*****************************************************************************
```

```
;*****************************************************************************
;*            PROTOTYPE ROUTE PLANNER FOR ZENITH 248
;*
;*    DATE: 18 August 1988
;*  VERSION: 1.0
;*   TITLE: Identify those files to be loaded during run time
;* FILENAME: autoload.s
;*
;* OPER SYS: DOS VERSION 3.2
;* LANGUAGE: PC Scheme (Ver 3.0 Student Edition) w/SCOOPS
;*  AUTHOR: Capt G.R. Gier
;*
;*  CONTENTS
;*      CLASSES: none
;*      METHODS: none
;*     COMMANDS: none
;*        TOOLS: none
;*    UTILITIES: none
;*    SYS/MACRO: autoload
;*
;* CALLED BY: Scheme
;*    CALLS: Any file listed
;*
;*   INPUTS: Not user accessible
;*  OUTPUTS: Invisible to user
;*
;*   VARIABLES USED: none
;* VARIABLES CHANGED: none
;*      FILES READ: Any fast-load file listed in this program
;*    FILES WRITTEN: none
;*
;*
;* FUNCTION:
;*
;* Scheme uses this file to interactively load blocks of code on demand.
;* Each line identifies the fastload file to load from (" .fsl") and
;* a list of the names of functions within the block. Some of the blocks
;* include method definitions that the function calls, but none of the blocks
;* contain only methods as there is no means to call them. The auto load
;* file saves computer memory as many of the tools below do not need to
;* be loaded unless one wants to inspect specific objects.
;*
;*****************************************************************************

(autoload-from-file "undup.fsl"   '(undup))
(autoload-from-file "flatten.fsl"  '(flatten))
(autoload-from-file "look.fsl"    '(look))
(autoload-from-file "life.fsl"    '(dead alive living?))
(autoload-from-file "vec_gt.fsl"  '(v>))
(autoload-from-file "vec_plus.fsl" '(v+))
(autoload-from-file "preempt.fsl"  '(preempt+))
(autoload-from-file "exclude.fsl"  '(exclude))
```

```
(autoload-from-file "tack_on.fsl"  '(tack-on))
(autoload-from-file "cost.fsl"     '(cost?))
(autoload-from-file "nodes.fsl"    '(nodes?))
(autoload-from-file "links.fsl"    '(links?))
(autoload-from-file "channels.fsl" '(channels?))
(autoload-from-file "circuits.fsl" '(circuits?))
(autoload-from-file "requests.fsl" '(requests?))
(autoload-from-file "replies.fsl"  '(replies?))
(autoload-from-file "mc.fsl"       '(mc))
(autoload-from-file "pr.fsl"       '(pr))
(autoload-from-file "out.fsl"      '(out))
(autoload-from-file "relay.fsl"    '(relay))
(autoload-from-file "sel_chan.fsl" '(select-channel))

;*****************************************************************************
;*                                                                           *
;*                              END                                          *
;*                                                                           *
;*****************************************************************************
```

```
;*****************************************************************************
;*          PROTOTYPE ROUTE PLANNER FOR ZENITH 248
;*
;*    DATE: 6 September 1988
;*  VERSION: 1.0
;*   TITLE: Macro for compiling all PRP files in their proper order
;* FILENAME: compile.me
;*
;* OPER SYS: DOS VERSION 3.2
;* LANGUAGE: PC Scheme (Ver 3.0 Student Edition) w/SCOOPS
;*  AUTHOR: Capt G.R. Gier
;*
;*  CONTENTS
;*     CLASSES: none
;*     METHODS: none
;*    COMMANDS: none
;*       TOOLS: none
;*    UTILITIES: none
;*    SYS/MACRO: compilation macro
;*
;* CALLED BY: User
;*    CALLS: none
;*
;*  INPUTS: none
;*  OUTPUTS: none
;*
;*   VARIABLES USED: none
;* VARIABLES CHANGED: none
;*      FILES READ: Source files with .s extention
;*    FILES WRITTEN: Object files with .so extention
;*
;*
;* FUNCTION:
;*
;* Works the same as if the programmer were to type in all the PRP files for
;* individual compilation.  Outputs OK when each file compilation is complete.
;*****************************************************************************

(load "c:scoops.fsl")
(compile-file "a:class.s"   "a:class.so")
   (display "class OK ")
(compile-file "a:draw.s"    "a:draw.so")
   (display "draw OK ")
(compile-file "a:flatten.s" "a:flatten.so")
   (display "flatten OK ")
(compile-file "a:undup.s"   "a:undup.so")
   (display "undup OK ")
(compile-file "a:look.s"    "a:look.so")
   (display "look OK ")
(compile-file "a:life.s"    "a:life.so")
   (display "life OK ")
(compile-file "a:vec_gt.s"  "a:vec_gt.so")
```

```
  (display "vector > OK ")
(compile-file "a:vec_plus.s" "a:vec_plus.so")
  (display "vector plus OK ")
(compile-file "a:preempt.s"  "a:preempt.so")
  (display "preempt OK ")
(compile-file "a:exclude.s"  "a:exclude.so")
  (display "exclude OK ")
(compile-file "a:tack_on.s"  "a:tack_on.so")
  (display "tack on OK ")
(compile-file "a:table_lu.s" "a:table_lu.so")
  (display "table lookup OK ")
(compile-file "a:cost.s"     "a:cost.so")
  (display "cost OK ")
(compile-file "a:out.s"      "a:out.so")
  (display "out OK ")
(compile-file "a:sel_chan.s" "a:sel_chan.so")
  (display "select channel OK ")
(compile-file "a:sort.s"     "a:sort.so")
  (display "sort OK ")
(compile-file "a:relay.s"    "a:relay.so")
  (display "relay OK ")
(compile-file "a:nodes.s"    "a:nodes.so")
  (display "nodes OK ")
(compile-file "a:links.s"    "a:links.so")
  (display "links OK ")
(compile-file "a:channels.s" "a:channels.so")
  (display "channel OK ")
(compile-file "a:circuits.s" "a:circuits.so")
  (display "circuits OK ")
(compile-file "a:requests.s" "a:requests.so")
  (display "requests OK ")
(compile-file "a:replies.s"  "a:replies.so")
  (display "replies OK ")
(compile-file "a:mc.s"       "a:mc.so")
  (display "make circuit OK ")
(compile-file "a:pr.s"       "a:pr.so")
  (display "print circuit OK ")
(compile-file "a:diamond.s"  "a:diamond.so")
  (display "diamond OK ")
(compile-file "a:stick.s"    "a:stick.so")
  (display "stick OK ")
(compile-file "a:star.s"     "a:star.so")
  (display "star OK ")
(compile-file "a:autoload.s" "a:autoload.so")
  (display "autoload OK ")

;*****************************************************************************
;*                                                                           *
;*                              END                                          *
;*                                                                           *
;*****************************************************************************
```

```
ECHO OFF
REM ****************************************************************************
REM              PROTOTYPE ROUTE PLANNER FOR ZENITH 248
REM
REM     DATE: 1 September 1988
REM   VERSION: 1.0
REM    TITLE: DOS batch file for making PRP fast-load files from object files
REM  FILENAME: fastload.bat
REM
REM  OPER SYS: DOS VERSION 3.2
REM  LANGUAGE: DOS
REM   AUTHOR: Capt G.R. Gier
REM
REM   CONTENTS
REM      CLASSES: none
REM      METHODS: none
REM     COMMANDS: none
REM        TOOLS: none
REM     UTILITIES: none
REM     SYS/MACRO: fastload batch file
REM
REM  CALLED BY: User
REM     CALLS: none
REM
REM   INPUTS: none
REM   OUTPUTS: none
REM
REM   VARIABLES USED: none
REM  VARIABLES CHANGED: none
REM      FILES READ: make_fsl.exe
REM    FILES WRITTEN: fastload files with an .fsl extention
REM
REM
REM  FUNCTION:
REM  This batch file makes multiple calls to make_fsl.exe so as to convert
REM  object files to fastload files using the Scheme utility make-fastload.
REM ****************************************************************************
ECHO ON
make_fsl   a:class.so   a:class.fsl
make_fsl    a:draw.so    a:draw.fsl
make_fsl  a:flatten.so  a:flatten.fsl
make_fsl   a:undup.so   a:undup.fsl
make_fsl    a:look.so    a:look.fsl
make_fsl    a:life.so    a:life.fsl
make_fsl  a:vec_gt.so  a:vec_gt.fsl
make_fsl a:vec_plus.so a:vec_plus.fsl
make_fsl  a:preempt.so  a:preempt.fsl
make_fsl  a:exclude.so  a:exclude.fsl
make_fsl  a:tack_on.so  a:tack_on.fsl
make_fsl a:table_lu.so a:table_lu.fsl
make_fsl    a:cost.so    a:cost.fsl
make_fsl     a:out.so     a:out.fsl
```

```
make_fsl a:sel_chan.so a:sel_chan.fsl
make_fsl    a:sort.so    a:sort.fsl
make_fsl   a:relay.so   a:relay.fsl
make_fsl   a:nodes.so   a:nodes.fsl
make_fsl   a:links.so   a:links.fsl
make_fsl a:channels.so a:channels.fsl
make_fsl a:circuits.so a:circuits.fsl
make_fsl a:requests.so a:requests.fsl
make_fsl  a:replies.so  a:replies.fsl
make_fsl      a:mc.so      a:mc.fsl
make_fsl      a:pr.so      a:pr.fsl
make_fsl  a:diamond.so  a:diamond.fsl
make_fsl   a:stick.so   a:stick.fsl
make_fsl    a:star.so    a:star.fsl
make_fsl a:autoload.so a:autoload.fsl

REM ******************************************************************************
REM                                                  *
REM                          END                       *
REM                                                  *
REM ******************************************************************************
```

```
;*****************************************************************************
;*          PROTOTYPE ROUTE PLANNER FOR ZENITH 248
;*
;*    DATE: 25 October 1988
;*  VERSION: 1.0
;*    TITLE: Scheme initialization files
;* FILENAME: scheme.ini
;*
;* OPER SYS: DOS VERSION 3.2
;* LANGUAGE: PC Scheme (Ver 3.0 Student Edition) w/SCOOPS
;*   AUTHOR: Capt G.R. Gier
;*
;*  CONTENTS
;*      CLASSES: none
;*      METHODS: none
;*     COMMANDS: none
;*        TOOLS: none
;*    UTILITIES: none
;*    SYS/MACRO: scheme initialization
;*
;* CALLED BY: User
;*    CALLS: none
;*
;*   INPUTS: none
;*  OUTPUTS: none
;*
;*   VARIABLES USED: none
;* VARIABLES CHANGED: none
;*      FILES READ: Those .fsl files listed
;*    FILES WRITTEN: none
;*
;*
;* FUNCTION:
;*
;* Calls fastload files in the proper order to execute PRP when PC Scheme is
;* first called.
;*****************************************************************************

(load "c:scoops.fsl")
(load   "class.fsl")
(load "autoload.fsl")
(load    "draw.fsl")
(load "table_lu.fsl")
(load    "sort.fsl")
(clear)

;*****************************************************************************
;*                                                                           *
;*                          END                                              *
;*                                                                           *
;*****************************************************************************
```

```
ECHO OFF
REM ***********************************************************************************
REM            PROTOTYPE ROUTE PLANNER FOR ZENITH 248
REM
REM     DATE: 9 August 1988
REM   VERSION: 1.0
REM    TITLE: Batch file for loading Scheme
REM  FILENAME: prp.bat
REM
REM  OPER SYS: DOS VERSION 3.2
REM  LANGUAGE: DOS
REM   AUTHOR: Capt G.R. Gier
REM
REM   CONTENTS
REM      CLASSES: none
REM      METHODS: none
REM     COMMANDS: none
REM       TOOLS: none
REM    UTILITIES: none
REM    SYS/MACRO: PRP preparatory batch file
REM
REM CALLED BY: User
REM    CALLS: none
REM
REM   INPUTS: none
REM   OUTPUTS: none
REM
REM   VARIABLES USED: none
REM VARIABLES CHANGED: none
REM      FILES READ: pcs.exe
REM    FILES WRITTEN: none
REM
REM
REM  FUNCTION:
REM
REM  This batch file changes path information so that Scheme can be read from
REM  the C: drive and PRP can be read from the A: drive.  Lastly it calls PC
REM  Scheme.
REM ***********************************************************************************
ECHO ON

cd c:\scheme
path = c:\scheme; a:\
cd a:
pcs
```

```
;*****************************************************************************
;*            PROTOTYPE ROUTE PLANNER FOR ZENITH 248
;*
;*    DATE: 25 August 1988
;*  VERSION: 1.0
;*   TITLE: Diamond shaped network file
;* FILENAME: diamond.s
;*
;* OPER SYS: DOS VERSION 3.2
;* LANGUAGE: PC Scheme (Ver 3.0 Student Edition) w/SCOOPS
;*  AUTHOR: Capt G.R. Gier
;*
;*  CONTENTS
;*      CLASSES: none
;*      METHODS: none
;*     COMMANDS: none
;*       TOOLS: none
;*    UTILITIES: none
;*    SYS/MACRO: none
;*      NETWORK: diamond
;*
;* CALLED BY: User
;*    CALLS: none
;*
;*  INPUTS: none
;*  OUTPUTS: Network objects
;*
;*  VARIABLES USED: *Circuit-ID*, Population
;* VARIABLES CHANGED: Population
;*      FILES READ: none
;*    FILES WRITTEN: none
;*
;*
;* FUNCTION:
;*
;* This block of code contains the commands to create a network of four
;* nodes, five links, 17 channels and two circuits.  It looks like:
;*                                  CNCE-1
;* Define nodes before            L-4 /|\ L-1   been created.  Use the
;* links, links before              / | \        message send format to
;* channels, and channels  CNCE-4 L-5|   CNCE-2   back connect previous
;* before circuits.  You            \ | /        objects.
;* can set instance variables     L-3 \|/ L-2
;* to objects that have already      CNCE-3
;*
;*****************************************************************************

(setcv NODE Population 0) ;Reset the node population upon loading.

(define *Circuit-ID* 2)   ;protect the two pre-created circuits.

(define CNCE-1
```

```
 (make-instance NODE
          'Name 'CNCE-1))
(define CNCE-2
 (make-instance NODE
          'Name 'CNCE-2))
(define CNCE-3
 (make-instance NODE
          'Name 'CNCE-3))
(define CNCE-4
 (make-instance NODE
          'Name 'CNCE-4))
(define LINK-1
 (make-instance LINK
          'Name 'LINK-1
          'Nodes (list CNCE-1 CNCE-2)))
(define LINK-2
 (make-instance LINK
          'Name 'LINK-2
          'Nodes (list CNCE-2 CNCE-3)))
(define LINK-3
 (make-instance LINK
          'Name 'LINK-3
          'Nodes (list CNCE-3 CNCE-4)))
(define LINK-4
 (make-instance LINK
          'Name 'LINK-4
          'Nodes (list CNCE-4 CNCE-1)))
(define LINK-5
 (make-instance LINK
          'Name 'LINK-5
          'Nodes (list CNCE-3 CNCE-1)))
(define CHANNEL-11
 (make-instance CHANNEL
          'Name 'CHANNEL-11
          'Link LINK-1))
(define CHANNEL-12
 (make-instance CHANNEL
          'Name 'CHANNEL-12
          'Link LINK-1))
(define CHANNEL-13
 (make-instance CHANNEL
          'Name 'CHANNEL-13
          'Link LINK-1))
(define CHANNEL-14
 (make-instance CHANNEL
          'Name 'CHANNEL-14
          'Link LINK-1))
(define CHANNEL-21
 (make-instance CHANNEL
          'Name 'CHANNEL-21
          'Link LINK-2))
(define CHANNEL-22
```

```
 (make-instance CHANNEL
          'Name 'CHANNEL-22
          'Link LINK-2))
(define CHANNEL-23
 (make-instance CHANNEL
          'Name 'CHANNEL-23
          'Link LINK-2))
(define CHANNEL-24
 (make-instance CHANNEL
          'Name 'CHANNEL-24
          'Link LINK-2))
(define CHANNEL-31
 (make-instance CHANNEL
          'Name 'CHANNEL-31
          'Link LINK-3))
(define CHANNEL-32
 (make-instance CHANNEL
          'Name 'CHANNEL-32
          'Link LINK-3))
(define CHANNEL-33
 (make-instance CHANNEL
          'Name 'CHANNEL-33
          'Link LINK-3))
(define CHANNEL-34
 (make-instance CHANNEL
          'Name 'CHANNEL-34
          'Link LINK-3))
(define CHANNEL-41
 (make-instance CHANNEL
          'Name 'CHANNEL-41
          'Link LINK-4))
(define CHANNEL-42
 (make-instance CHANNEL
          'Name 'CHANNEL-42
          'Link LINK-4))
(define CHANNEL-43
 (make-instance CHANNEL
          'Name 'CHANNEL-43
          'Link LINK-4))
(define CHANNEL-44
 (make-instance CHANNEL
          'Name 'CHANNEL-44
          'Link LINK-4))
(define CHANNEL-51
 (make-instance CHANNEL
          'Name 'CHANNEL-51
          'Link LINK-5))
(define CIRCUIT-1
 (make-instance CIRCUIT
          'Name 'CIRCUIT-1
          'Priority-Cost (vector 0 0 1 0)))
(define CIRCUIT-2
```

```
(make-instance CIRCUIT
        'Name 'CIRCUIT-2
        'Priority-Cost (vector 0 1 0 0))) -
(send CNCE-1 set-Links (list LINK-1 LINK-4 LINK-5))
(send CNCE-2 set-Links (list LINK-1 LINK-2))
(send CNCE-3 set-Links (list LINK-2 LINK-3 LINK-5))
(send CNCE-4 set-Links (list LINK-3 LINK-4))
(send LINK-1 set-Channels (list CHANNEL-11 CHANNEL-12 CHANNEL-13 CHANNEL-14))
(send LINK-2 set-Channels (list CHANNEL-21 CHANNEL-22 CHANNEL-23 CHANNEL-24))
(send LINK-3 set-Channels (list CHANNEL-31 CHANNEL-32 CHANNEL-33 CHANNEL-34))
(send LINK-4 set-Channels (list CHANNEL-41 CHANNEL-42 CHANNEL-43 CHANNEL-44))
(send LINK-5 set-Channels (list CHANNEL-51))
(send CHANNEL-11 set-circuit circuit-1)
(send CHANNEL-21 set-circuit circuit-1)
(send CHANNEL-31 set-circuit circuit-2)

;*****************************************************************************
;*                                                                           *
;*                          END                                              *
;*                                                                           *
;*****************************************************************************
```

```
;*****************************************************************************
;*          PROTOTYPE ROUTE PLANNER FOR ZENITH 248
;*
;*    DATE: 27 October 1988
;*  VERSION: 1.0
;*   TITLE: Star shaped network file
;* FILENAME: star.s
;*
;* OPER SYS: DOS VERSION 3.2
;* LANGUAGE: PC Scheme (Ver 3.0 Student Edition) w/SCOOPS
;*  AUTHOR: Capt G.R. Gier
;*
;*  CONTENTS
;*     CLASSES: none
;*     METHODS: none
;*    COMMANDS: none
;*       TOOLS: none
;*    UTILITIES: none
;*    SYS/MACRO: none
;*     NETWORK: star
;*
;* CALLED BY: User
;*    CALLS: none
;*
;*  INPUTS: none
;*  OUTPUTS: Network objects
;*
;*  VARIABLES USED: *Circuit-ID*, Population
;* VARIABLES CHANGED: Population
;*     FILES READ: none
;*    FILES WRITTEN: none
;*
;*
;* FUNCTION:
;*
;* This block of code contains the commands to create a network of
;* five nodes, five links, 20 channels and no circuits.  It looks like:
;*                                  CNCE-1
;* Define nodes before            L-3 /\L-1        Use the message
;* links, links bef              CNCE-5--/-\--CNCE-2  message send format to
;* channels, ... channels     L-2' <    > 'L-4     back connect previous
;* before circuits.                 / V \          objects.
;*                               L-5/ / \ \
;*                              CNCE-4   CNCE-3
;*
;*****************************************************************************

(setcv NODE Population 0)  ;Reset upon loading

(define *CIRCUIT-ID* 0)

(define CNCE-1
```

```
 (make-instance NODE
          'Name 'CNCE-1))
(define CNCE-2
 (make-instance NODE
          'Name 'CNCE-2))
(define CNCE-3
 (make-instance NODE
          'Name 'CNCE-3))
(define CNCE-4
 (make-instance NODE
          'Name 'CNCE-4))
(define CNCE-5
 (make-instance NODE
          'Name 'CNCE-5))
(define LINK-1
 (make-instance LINK
          'Name 'LINK-1
          'Nodes (list CNCE-1 CNCE-3)))
(define LINK-2
 (make-instance LINK
          'Name 'LINK-2
          'Nodes (list CNCE-3 CNCE-5)))
(define LINK-3
 (make-instance LINK
          'Name 'LINK-3
          'Nodes (list CNCE-5 CNCE-2)))
(define LINK-4
 (make-instance LINK
          'Name 'LINK-4
          'Nodes (list CNCE-2 CNCE-4)))
(define LINK-5
 (make-instance LINK
          'Name 'LINK-5
          'Nodes (list CNCE-4 CNCE-1)))
(define CHANNEL-11
 (make-instance CHANNEL
          'Name 'CHANNEL-11
          'Link LINK-1))
(define CHANNEL-12
 (make-instance CHANNEL
          'Name 'CHANNEL-12
          'Link LINK-1))
(define CHANNEL-13
 (make-instance CHANNEL
          'Name 'CHANNEL-13
          'Link LINK-1))
(define CHANNEL-14
 (make-instance CHANNEL
          'Name 'CHANNEL-14
          'Link LINK-1))
(define CHANNEL-21
 (make-instance CHANNEL
```

```
            'Name 'CHANNEL-21
            'Link LINK-2))
(define CHANNEL-22
 (make-instance CHANNEL
            'Name 'CHANNEL-22
            'Link LINK-2))
(define CHANNEL-23
 (make-instance CHANNEL
            'Name 'CHANNEL-23
            'Link LINK-2))
(define CHANNEL-24
 (make-instance CHANNEL
            'Name 'CHANNEL-24
            'Link LINK-2))
(define CHANNEL-31
 (make-instance CHANNEL
            'Name 'CHANNEL-31
            'Link LINK-3))
(define CHANNEL-32
 (make-instance CHANNEL
            'Name 'CHANNEL-32
            'Link LINK-3))
(define CHANNEL-33
 (make-instance CHANNEL
            'Name 'CHANNEL-33
            'Link LINK-3))
(define CHANNEL-34
 (make-instance CHANNEL
            'Name 'CHANNEL-34
            'Link LINK-3))
(define CHANNEL-41
 (make-instance CHANNEL
            'Name 'CHANNEL-41
            'Link LINK-4))
(define CHANNEL-42
 (make-instance CHANNEL
            'Name 'CHANNEL-42
            'Link LINK-4))
(define CHANNEL-43
 (make-instance CHANNEL
            'Name 'CHANNEL-43
            'Link LINK-4))
(define CHANNEL-44
 (make-instance CHANNEL
            'Name 'CHANNEL-44
            'Link LINK-4))
(define CHANNEL-51
 (make-instance CHANNEL
            'Name 'CHANNEL-51
            'Link LINK-5))
(define CHANNEL-52
 (make-instance CHANNEL
```

```
          'Name 'CHANNEL-52
          'Link LINK-5))
(define CHANNEL-53
 (make-instance CHANNEL
          'Name 'CHANNEL-53
          'Link LINK-5))
(define CHANNEL-54
 (make-instance CHANNEL
          'Name 'CHANNEL-54
          'Link LINK-5))
(send CNCE-1 set-Links (list LINK-1 LINK-5))
(send CNCE-2 set-Links (list LINK-3 LINK-4))
(send CNCE-3 set-Links (list LINK-1 LINK-2))
(send CNCE-4 set-Links (list LINK-4 LINK-5))
(send CNCE-5 set-Links (list LINK-2 LINK-3))
(send LINK-1 set-Channels (list CHANNEL-11 CHANNEL-12 CHANNEL-13 CHANNEL-14))
(send LINK-2 set-Channels (list CHANNEL-21 CHANNEL-22 CHANNEL-23 CHANNEL-24))
(send LINK-3 set-Channels (list CHANNEL-31 CHANNEL-32 CHANNEL-33 CHANNEL-34))
(send LINK-4 set-Channels (list CHANNEL-41 CHANNEL-42 CHANNEL-43 CHANNEL-44))
(send LINK-5 set-Channels (list CHANNEL-51 CHANNEL-52 CHANNEL-53 CHANNEL-54))

;*****************************************************************************
;*                                                                           *
;*                              END                                          *
;*                                                                           *
;*****************************************************************************
```

```
;*****************************************************************************
;*            PROTOTYPE ROUTE PLANNER FOR ZENITH 248
;*
;*    DATE: 24 August 1988
;*  VERSION: 1.0
;*    TITLE: Single path network file
;* FILENAME: stick.s
;*
;* OPER SYS: DOS VERSION 3.2
;* LANGUAGE: PC Scheme (Ver 3.0 Student Edition) w/SCOOPS
;*   AUTHOR: Capt G.R. Gier
;*
;*  CONTENTS
;*      CLASSES: none
;*      METHODS: none
;*     COMMANDS: none
;*        TOOLS: none
;*    UTILITIES: none
;*    SYS/MACRO: none
;*      NETWORK: stick
;*
;* CALLED BY: User
;*    CALLS: none
;*
;*   INPUTS: none
;*  OUTPUTS: Network objects
;*
;*   VARIABLES USED: *Circuit-ID*, Population
;* VARIABLES CHANGED: Population
;*      FILES READ: none
;*    FILES WRITTEN: none
;*
;*
;* FUNCTION:
;*
;* This simple network works well when debugging new code as it has few
;* components, yet contains all object classes except circuits.
;*
;*              CNCE-1———CNCE-2———CNCE-3
;*                 L-1       L-2
;*                (Ch-1)    (Ch-2)
;*
;* Notice that fewer messages can be sent if subsequent definitions contain
;* instance variables set to previously defined objects.  Messages serve to
;* back-connect those objects that do not exist at the time of instantiation.
;*
;*****************************************************************************

(setcv NODE Population 0)   ;Reset node population

(define *Circuit-ID* 0)
```

```
(define CNCE-1
 (make-instance NODE
          'Name 'CNCE-1))
(define CNCE-2
 (make-instance NODE
          'Name 'CNCE-2))
(define CNCE-3
 (make-instance NODE
          'Name 'CNCE-3))
(define LINK-1
 (make-instance LINK
          'Name 'LINK-1
          'Nodes (list CNCE-1 CNCE-2)))
(define LINK-2
 (make-instance LINK
          'Name 'LINK-2
          'Nodes (list CNCE-2 CNCE-3)))
(define CHANNEL-1
 (make-instance CHANNEL
          'Name 'CHANNEL-1
          'Link LINK-1))
(define CHANNEL-2
 (make-instance CHANNEL
          'Name 'CHANNEL-2
          'Link LINK-2))
(send CNCE-1 set-Links (list LINK-1))
(send CNCE-2 set-Links (list LINK-1 LINK-2))
(send CNCE-3 set-Links (list LINK-2))
(send LINK-1 set-Channels (list CHANNEL-1))
(send LINK-2 set-Channels (list CHANNEL-2))

;*****************************************************************************
;*                                                                           *
;*                          END                                              *
;*                                                                           *
;*****************************************************************************
```

Bibliography

Aho, Alfred V. and others. The Design and Analysis of Computer Algorithms. Reading MA: Addison-Wesley Publishing Company, Inc., 1974.

Booch, Grady. Software Components With Ada. Menlo Park CA: Benjamin/Cummings Publishing Company, Inc., 1987.

Christofides, Nicos. Graph Theory, an Algorithmic Approach. London: Academic Press, Inc., 1975.

Computer Sciences Corporation. DPAS Network Control System Final Technical Report, Draft. Prepared for: Department of the Air Force, Hqs, Rome Air Development Center (AFSC), Griffiss Air Force Base, New York 13441. Prepared under contract F30602-86-C-0246, May 1988.

Conry, S. E. et al. "Multistage Negotiation in Distributed Planning," Unpublished paper under contract No. F30602-85-C-0008. Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base NY, 15 December 1986.

Davis, Ernest. "Constraint Propagation with Interval Labels," Artificial Intelligence, an International Journal, 32: 281-331 (July, 1987).

Ludwig, Gunter and Robert Roy. "Saturation Routing Network Limits," Proceedings of the IEEE, 65: 1353-1362 (September 1977).

Markland, Robert, E. Topics in Management Science. New York: John Wiley and Sons, 1983.

Minieka, Edward. Optimization Algorithms for Networks and Graphs. New York: Marcel Dekker, Inc., 1978.

Rich, Elaine. Artificial Intelligence. New York: McGraw-Hill Book Company, 1983.

Stallman, Richard M. and Gerald J. Sussman. "Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis" Artificial Intelligence, an International Journal, 9: 135-196 (October 1977).

Sues, Larry, Acting thesis sponsor. Telephone interview. RADC/DCLD, Griffiss AFB NY, 20 October 1988.

Texas Instruments. PC Scheme, a Simple, Modern LISP. User's Guide, Revision B. Texas Instruments Incorporated, Data Systems Group, July 1987.

Warmuth, Lt Col Donald, Former CNCE Program Manager. Personal interview. ESD/TCR, Hanscom AFB MA, 15 April 1988.

Vita

Captain Glenn Richard Gier [redacted] in 1974 [redacted] attended the United States Air Force Academy, from which he received the degree of Bachelor of Science in Electrical Engineering and his commission in May 1978. He completed pilot training in October 1979 from Williams AFB, AZ, then served as a C-141 pilot scheduler, flight instructor, and base executive officer while assigned to the 15th Military Airlift Squadron, Norton AFB, CA. He attended Squadron Officers School, May 1985, and in August that same year was assigned to the 4950th Test Squadron, Wright-Patterson AFB, OH, were he was a project pilot, flight instructor, and Silent Attack Warning System project manager. In June, 1987 Captain Gier entered the School of Engineering, Air Force Institute of Technology and is currently working on the degree of Masters of Science in Electrical Engineering.

[redacted]

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; Distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GE/ENG/88D-13	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering 6b. OFFICE SYMBOL (If applicable) AFIT/ENA	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) Air Force Institute of Technology Wright-Patterson AFB. Ohio 45433-6583	7b. ADDRESS (City, State, and ZIP Code) B Franiewicki 12 Jan 1989
8a. NAME OF FUNDING/SPONSORING ORGANIZATION RADC 8b. OFFICE SYMBOL (If applicable) DCLD	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) Griffiss AFB, NY 13411-5200	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
OBJECT-ORIENTED ALLOCATION OF RESOURCES IN A TACTICAL COMMUNICATIONS NETWORK (UNCLASSIFIED)

12. PERSONAL AUTHOR(S)
Glenn R. Gier, Capt, USAF

13a. TYPE OF REPORT	13b TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
MS Thesis	FROM _____ TO _____	1988 December	138

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Tactical Communication, Communications Networks
12	07		Decision Making, Digital Simulation
25	02		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Thesis Chairman: Dr. Frank M. Brown
Professor of Electrical Engineering

Abstract:
The circuits in a military command and control network are expected to operate continuously in spite of changes and damage, and must be restored in minutes should they fail. This study examines circuit-building as a resource allocation problem, and describes an approach to the decentralized allocation of prioritized circuits in such a network. Based on a saturation search algorithm, a circuit-request message ripples from an originating node through the network to the circuit's destination node, providing a path exists. Along the way the message retains cost and path information that is used to attenuate expensive routes and provide path information during the back-sweep of the allocation phase.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Dr. Frank M. Brown	22b. TELEPHONE (Include Area Code) (513) 255-3450	22c. OFFICE SYMBOL AFIT/ENG

DD Form 1473, JUN 86 Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Abstract, continued:
An object-oriented program written in Scheme helped capture the details of nodes, links, channels, and circuits in the network, and showed how these components would interact when guided by the algorithm. The program has two purposes. First, it validates the concept that a common software-based assistant, distributed to each node, can aid operators in the rapid reconstruction of a badly damaged network. Second, it provides a planning aide for tactical network designers, who can use the program to model nodes, trunk-lines, channels and circuits.

A tactical network employing distributed allocation and priority-coding of its circuits, allows operators to forgo stored restoration plans as their principal means of maintaining service. This approach offers flexibility and responsiveness despite the likelihood of multiple outages and rapid changes, something that plans can not always deliver especially in time of war.